UNIVERSITAT POLITÈCNICA
DE CATALUNYA

Programa de Doctorat en Computació
Departament de Llenguatges i Sistemes
Informàtics
Universitat Politècnica de Catalunya

IMPROVING AUTOMATIC RIGGING FOR 3D HUMANOID CHARACTERS

Jorge Eduardo Ramírez Flores

Tesi presentada per obtenir el títol de
Doctor per la Universitat Politècnica de
Catalunya

Advisor: Antonio Susín Sánchez

Barcelona 2015

Abstract.

In the field of computer animation the process of creating an animated character is usually a long and tedious task. An animation character is usually defined by a $3D$ mesh (a set of triangles in the space) that gives its external appearance or shape to the character. It also used to have an inner structure, the skeleton. When a skeleton is associated to a character mesh, this association is called skeleton binding, and a skeleton bound to a character mesh is an animation rig.

Rigging from scratch a character can be a very boring process. The definition and creation of a centered skeleton together with the 'painting', by an artist, of the influence parameters between the skeleton and the mesh (the skinning) is the most demanding part to achieve an acceptable behavior for a character. This rigging process can be simplified and accelerated using an automatic rigging method. Automatic rigging methods consist in taking as input a 3D mesh, generate a skeleton based in the shape of the original model, bound the input mesh to the generated skeleton, and finally to compute a set of parameters based in a chosen skinning method. The main objective of this thesis is to generate a method for rigging a $3D$ arbitrary model with minimum user interaction. This can be useful to people without experience in the animation field or to experienced people to accelerate the rigging process from days to hours or minutes depending the needed quality. Having in mind this situation we have designed our method as a set of tools that can be applied to general input models defined by an artist. The contributions made in the development of this thesis can be summarized as:

- **Generation of an animation Rig (Chap. 2):** Having an arbitrary closed mesh we have implemented a thinning method to create first an unrefined geometry skeleton that captures the topology and pose of the input character. Using this geometric skeleton as starting point we use a refining method that creates an adjusted logic skeleton based in a template, or may be defined by the user, that is compatible with the current animation formats. The output logic skeleton is specific for each character, and it is bounded to the input mesh to create an animation rig.

- **Skinning (Chap. 3 and 4):** Having defined an animation rig for an arbitrary mesh we have developed an improved skinning method;

this method is based on the Linear Blend Skinning(LBS) algorithm. Our contributions in the skinning field can be sub-divided in:

  - We propose a segmentation method (Chap. 3) that works as the core element in a weight assigning algorithm and a skinning algorithm, we also have developed an automatic algorithm to compute the skin weights of the LBS Skinning of a rigged polygonal mesh (Sec. 4.2).
  - Our proposed skinning algorithm uses as base the features of the LBS Skinning (Sec. 4.4). The main purpose of the developed algorithm is to solve the well-known "candy wrap" artifact; that produces a substantial loss of volume when a link of an animation skeleton is rotated over its own axis. We have compared our results with the most important methods in the skinning field, such as Dual Quaternion Skinning (DQS) and LBS, achieving a better performance over DQS and an improvement in quality over LBS.

- **Animation tools (Chap. 5):** We have developed a set of Autodesk Maya commands that works together as rig tool, using our previous proposed methods.
- **Animation loader (Sec. 5.1):** Moreover, an animation loader tool has been implemented, that allows the user to load animations from a skeleton with different structure to a rigged 3D model.

The contributions previously described has been published in 3 research papers, the first two were presented in international congresses and the third one was acepted for its publication in an JCR indexed journal.

Agradecimientos.

El terminar esta tesis es la conclusión de un proyecto y un sueño el cual ha requerido mucho esfuerzo y dedicación de parte de todas las personas que estuvieron involucradas, definitivamente el haber estudiado un Doctorado ha sido para mí un evento que me ha cambiado totalmente; espero para bien. Sinceramente creo que hay un antes y un después en lo que a mi persona se refiere, en el desarrollo de esta tesis he conocido lo dura que es la frustración y lo difícil que es a veces el seguir persiguiendo tus sueños, lo que yo llamo: "el ir detrás del conejo blanco".

Creo tal vez, que comencé a estudiar el doctorado con la idea equivocada, me perdí muchas veces (demasiadas, muy a mi pesar), y cuando al fin me centré en lo que realmente importaba fue cuando las cosas comenzaron a funcionar, ¿por qué fue así?, no lo sé; pero creo que era lo que necesitaba para madurar. Si el proceso fue doloroso es porque siempre he sido duro de cabeza y muy terco... al final creo que tu mejor virtud también termina siendo tu peor defecto. De esta experiencia puedo decir una cosa: me encanta jugar con una computadora, y me pierde adentrarme en este tipo de temas, espero que la vida me siga brindando la oportunidad de seguir divirtiéndome.

Me gustaría agradecer a todos mis amigos, los que estuvieron conmigo, me alentaron y me escucharon aun en mis momentos más obscuros: Abraham de la Rosa, Benjamín Hernández Arreguín, Marco Lino Calderón, Erick Arechiga, Jeniffer Strauss, Xavi Lligadas y tantos otros que se me vienen a la mente pero por falta de espacio no menciono.

A mi asesor: Antonio Susín Sánchez, por su guía y paciencia, sin él no hubiese sido capaz de concluir este objetivo.

Al Consejo Nacional y Ciencia y Tecnología (CONACYT) por su apoyo invaluable en el desarrollo de esta tesis.

Y finalmente pero en primer lugar de importancia, a mi familia: mi hermano y su hermosa familia, y a mis padres, muy en especial a mis padres. Les pido mil disculpas por arrastrarlos junto conmigo a algo como esto, creo que mis decisiones no son a veces las mejores y ustedes han tenido que soportar las consecuencias junto conmigo cuando en realidad no les tocaba; si hoy o algún día llego a valer la pena como ser humano es todo gracias a ustedes.

Son el mayor ejemplo de lealtad, honradez y dignidad que cualquier hijo pudiera querer. No sé como agradecerles todo lo que han hecho por mí en la vida y creo que nunca lo lograré, esta tesis no es un triunfo mío, es de ustedes.

Contents

List of Figures

List of Tables

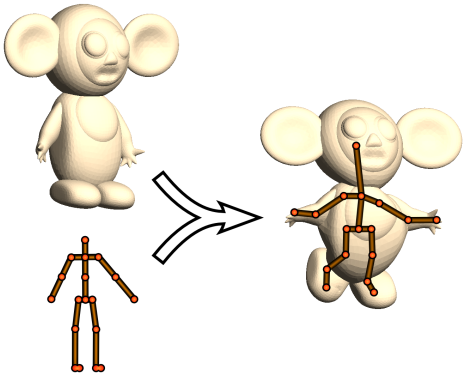
List of Algorithms

Chapter 1

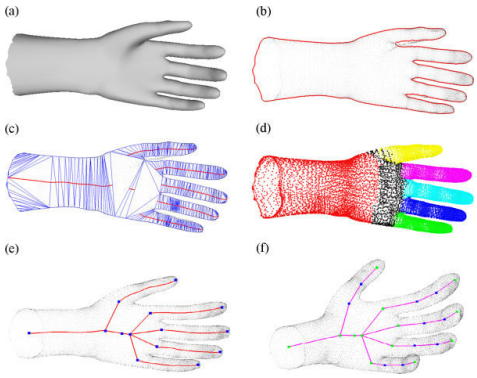
Introduction.

In the field of computer animation, the process of creating an animated character is usually long and tedious. A character in computer animation is a $3D$ polygonal mesh that is "sculpted" by a graphic artist; the sculpting process can be made by using a variety of techniques like: Box/Subdivision Modeling, Edge/Contour Modeling, Nurbs/Spiline Modeling, Digital Sculpting, Procedural Modeling, Image Based Modeling and $3D$ Scanning. As an example, Box/Subdivision Modeling consists in taking a geometric object as a base (a cube, tetrahedron, sphere, etc.), modify it, add or join its vertices, and refine them in their surface until the desired shape is achieved. A brief recapitulation of all the mentioned methods can be found in [41]. The animation of a sculpted character can be achieved in different ways: *cage based* and *skeleton driven*.
In the case of *skeleton-drive* animation, a skeleton is defined inside of the character's $3D$ mesh. A skeleton is a set of points that represent the limbs of a character and is defined through joints and links. A link is the equivalent to a bone, and each link consists of two joints, which are points in the space that define the local space where rigid transformations (rotations and translations) are performed. The positions of the joints are computed in a hierarchy; therefore, any rigid transformation over a joint will impact all the joints that are in a lower hierarchical level. The association of a skeleton to a character mesh is called skeleton binding, and a skeleton bound to a character mesh is known as an animation rig. A set of rigid transformations in a specific amount of time over the joint's character are known as an animation sequence. Once the input model was rigged, a set of influence parameters are painted by a digital artist to define the way that a limb must deform.
Rigging a character can be a tedious process, as has been mentioned in [3], [8] and [30], the definition and creation of a centered skeleton is time con-

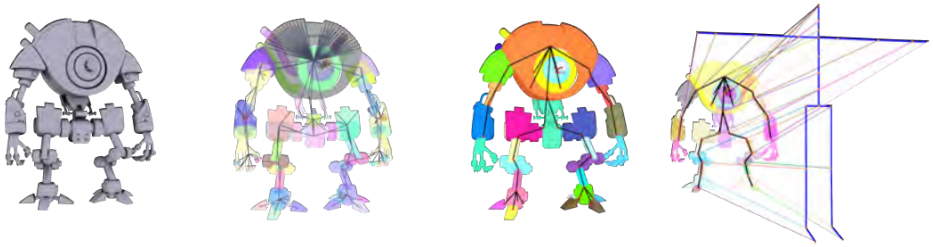
suming; but the process of painting the influence parameters by an artist is the most demanding part in the process to achieve an acceptable limb behavior for a character. The rigging process can be simplified and accelerated using an *automatic rigging* method. These automatic rigging methods take a character's $3D$ mesh as an input, a skeleton is then generated based on the shape of the original model. After the skeleton is created, the input mesh is bounded to it, and finally a set of parameters based on a chosen skinning method are computed. The most relevant work in this field is the method proposed by Baran and Popovic [3] called **Pinocchio**. **Pinocchio** takes a closed mesh as an input, and creates a rig though skeleton embedding; *Linear Blend Skinning(LBS)* is used as a deformation method for the character's animation.



(a) Pinocchio automatic rig (image taken from [3]).



(b) Automatic rigging with $3D$ silhouette (image taken from [34]).



(c) Automatically Rigging Multi-component Characters [4].

Figure 1.1: *Automatic rigging methods.*

An automatic rigging method can be subdivided in the next sub-problems: skeleton extraction, skeleton adjusting, skinning algorithm parameters and animation transfer. Each one of these "sub-problems" are interesting and important on their own. Skeleton extraction is a problem that has been

studied in medical imaging, pattern recognition, scientific visualization and CAD. Algorithms related with skeleton extraction, such as thinning, work with solid voxelized models [33] and medial axis extraction from vertices clouds [43]. The mentioned algorithms were developed to be applied in a different research field than computer animation but they can be applied to solve one of the previous sub-problems.

An extracted skeleton is a set of points in space with no additional information; to be useful in an animation context, the extracted skeleton must be transformed into a connected inverse kinematic graph. Methods like [34], [48] and [2] create an animation skeleton from a set of rules or a defined algorithm which produces a rigged model without an animation file associated; therefore, an animation sequence must be adapted or made specifically for the extracted skeleton.

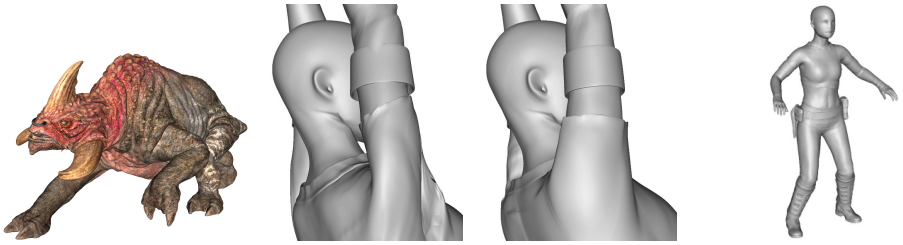
Creating or adapting an animation sequence to a rigged model is not easy. The created skeleton can be more complex (having more joints than a motion capture skeleton) than the one defined in the animation source, therefore complicating the task even more. That is the reason why we believe that a good automatic rigging method should have a skeleton adjust stage, or an equivalent method. An interesting approach is the one taken in [3]; where a skeleton with animation information to be embedded was used (instead of adjusting an extracted skeleton) on a $3D$ model with similar shape and posed as the input model.

A more traditional process was made in [4], where after extracting a skeleton from an arbitrary mesh with similar shape and pose, an equivalent animation skeleton was created and refined by a joint re-targeting process from the chosen animation skeleton, completing the skeleton adjust stage.

The rigging process is inevitably attached to a deformation method; the chosen deformation method is the engine part that allows us to move the input $3D$ model along with the extracted skeleton. In computer animation, a deformation method bounded to a skeleton is commonly called skinning. Skinning the most popular and researched of the three tasks listed, the meaning of skinning in computer animation is an algorithm that *deforms* or changes the position of the vertices on a $3D$ polygonal mesh; therefore a skinning algorithm is a function that takes as input vertices and changes their position based on a set of arguments and external factors. The function nature can be linear (linear blend skinning (LBS) is the classic linear algorithm in skeleton driven animation) or non-linear (mean value coordinates (MVC) and dual quaternion skinning (DQS) are good examples of non-linear algorithms in cage and skeleton driven animation respectively). The external factors can be a set of points (control points in a cage based deformation scheme) outside the input model, or a set of points inside the

mesh (points that form a skeleton in skeleton driven animation scheme).

(a) Harmonic coordinates used by Pixar. (b) *HC* used in Ratatouille. (c) Cage used in *HC*.

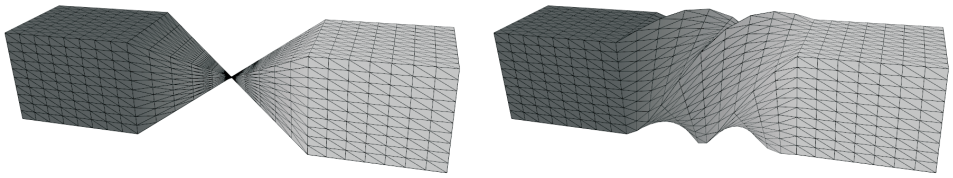


(d) *MWE*. (e) *MWE*. (f) *MWE*. (g) *MWE*.

Figure 1.2: *Skinning methods used in film industry, Harmonic coordinates from* ***Pixar*** *(images* ***a-c*** *taken from [15] and [7]) and Multi-Weight Enveloping from* ***Industrial Light and Magic*** *(images* ***d-g*** *taken from [46]).*

On film and video game industries, skinning is used extensively; therefore is one of their main research topics, although the focus from each one is different. Film industry puts an emphasis on realism, leaving aside computation times. Game industry is the opposite, focusing their efforts in a fast and efficient algorithm. That difference of perspectives is the reason behind the variety of skinning algorithms, animated film industry uses mostly free form deformation (*FFD*) or cage based skinning methods; such as *Pixar* with their *harmonic coordinates [7]*, that allow the animator to have control over the deformed parts to create some cartoon-like effects. *FFD* and cage based skinning uses control points instead of a *skeleton* to control the volume inside of a defined cage. In film industry, cage based skinning and *FFD* are not the only alternatives, an expansion of *LBS* called Multi-Weight Enveloping [46] was used by *Industrial Light and Magic* in the episode II of star wars saga; an interesting alternative that combines the two skinning methods has emerged like [17]. Where the authors use cage

based skinning along with an animation skeleton using a set of templates with the objective of creating a cage that fulfills the same function as a skeleton. The cage is deformed with a modification of the *LBS* and the mesh inside the cages are deformed with *positive mean value coordinates*. In game industry, the most used method of skinning is the efficient *LBS*; all the graphic engines in the industry supports it, such as *Unreal Engine* that its native skinning algorithm is *LBS*. If another skinning algorithm is going to be used, it has to be developed as an extra library (at least that is the case in *Unreal Engine 3*); but recently *Crytek* has introduced *Dual Quaternion Skinning* in their *CryENGINE 3*, using one of the most sophisticated non-linear skinning method nowadays. *Dual Quaternion Skinning* can use the same weights parameters as *LBS*, but to improve the quality of the deformations in the model, the rigging and the weight painting has to be done exclusively for *DQS*.



(a) *LBS* candy wrapper artifact (image taken from [20]).

(b) *DQS* with the same twist rotation as *LBS* (image taken from [20]).

Figure 1.3: *Candy wrapper artifact in* ***LBS****, same rotation solved properly with* ***DQS****.*

Motivation.

The motivation behind the elaboration of this thesis is to automatize the rigging process of an animation character. To achieve this goal, we are going to divide the pipeline of the rigging process and treat them as individual problems, developing tools and solutions for each part. Our set of tools will be designed to help people with low knowledge about animation; but we also want to create a set of tools that can be used by experienced users, such as digital artist, to use it as a base to their work. Our methods can also be used to obtain a preliminary version of a rigged character in the fields of research, video game development and those affine to animation that needs a fast preliminary result.

Objectives.

The main objective of this thesis, is to generate a rig method for arbitrary $3D$ models that need a minimum interaction from the user. Our method has to be useful to people without experience in the animation field, and at the same time be useful to experienced users that wants to accelerate the rigging process from days to hours or even minutes, depending the needed quality. This objective is ambitious, because inexperienced users ignore most of the core elements in the process, and on the other hand, we have experts that want all the control over the rigging process. Having in mind this situation, we have designed our method as a set of tools that can be applied separately to an input model but with a specific sequence to rig a model. For unexperienced users, we have a predefined skeleton that is used as a template to generate a rig from a closed $3D$ model. Expert users can change this template with a more suitable skeleton that fulfills their needs, or modify the output of the rig tool to improve the position of the created joints that could not be achieved by an automatic algorithm. Along with the rigging tool, other of the objectives is to create a tool to assign the influence weights automatically for the widely used *LBS* deformation scheme. The common approach taken to produce the influence weights automatically is: treat the weight assignment as a minimization problem, using a high-order function as the base to create smooth transitions between influence joints. On the contrary we want to propose a segmentation algorithm to deal with the weight assign problem based on segmentation. We believe that a segmented model simplifies and makes easier the weight assignment problem; simplifying the problem to a diffused one between a reduced number of influence joints, that can be solved using a normalized diffusion function. Our weight assigning tool will be designed to be applied independently of the rig tool; with a rigged mesh as the input that can be created with our rig tool, or manually.

The *LBS* algorithm is known for being the most efficient of the deformation algorithms, but it's also known for producing candy wrapper artifact (a collapse of the model's volume around a joint when a twist rotation is performed). We want to propose an alternative to the *LBS* algorithm that has the main feature of being fully compatible with the influence weights used for the *LBS* algorithm, but without their known artifacts and with the characteristic of being a separated module that can be used when more quality is needed instead of the *LBS* deformation scheme.
A rigged mesh is not useful if it is not associated to any animation file. In the case of artists in the animation industry, there is no problem with getting animations for a specific rig, but users without this possibility will

have to edit animation files by hand or find an option in commercial software. Having this in mind, we want to complete our set of tools with an animation loader tool that will find equivalences between two animation skeletons and transfer and adapt the rigid transformations (rotations and translations) from a skeleton defined by an animation file to a rigged model. All the developed software in this thesis will be created as a plug-in to one of the major animation software packages available in the industry: *Autodesk Maya*. This will allow the final user to combine our set of tools with all the possibilities that offers one of the best animation package ever created.

To achieve our objective of creating a set of tools to rig a $3D$ model, we will summarize the contributions of this thesis as:

- Generation of an animation Rig: Having an arbitrary mesh (a $3D$ sculpted character), we want to automatize most of the process by achieving the next sub-objectives:

    1. To implement a voxelizing and thinning algorithm over an input $3D$ model, to create an *unrefined* geometry skeleton. (sections 2.1 and 2.1)
    2. To classify the kinds of nodes in an unrefined skeleton and design a transversal skeleton algorithm that transforms the geometric nodes into a data structure This data structure will make the analysis of the position, neighborhood and dimensions of the skeleton branches easyer(sections 2.2 and 2.3).
    3. To propose a method to create a geometric skeleton that captures the topology, dimension and pose of the character mesh. Refining the extracted skeleton by trimming unnecessary branches; using the labeled end nodes by a human user as base of our method's reffining process(section 2.3).
    4. To generate an adjusted logic skeleton for the polygonal mesh using one defined by a user or taking a predefined template. The adjustment of the logic skeleton will be made using a geometric skeleton (section 2.4).

- Skinning: Having defined an animation rig for an arbitrary mesh we will develop a skinning method which will be based on the *Linear Blend Skinning* algorithm. Our objective in the skinning field can be sub-divided in:

    1. To propose an innovative segmentation method that works as the core element in a weight assigning algorithm and skinning algorithm. Two kinds of algorithms will be developed: one mainly

designed to work with models in an ideal *T-pose* based entirely on the geometry of the input model; and other two that work with models in arbitrary poses: one based on a voxelized version of the input model and other that will be geometrically based (section3.2).

2. To create an automatic algorithm to compute the *weights* of the **LBS** for a rigged polygonal mesh. The algorithm will have the main purpose of being used by people with low knowledge about skinning, or the initial approximation of a professional work used by an experienced user (section 4.2).

3. To build a skinning algorithm that have as its base the features of the **LBS** Skinning. The main objective of the developed algorithm is to solve the well-known "candy wrap" artifact created when a twist rotation is performed over one of the joints of the character (section 4.4).

- To show and explain the set of created *Autodesk Maya* commands that work together as a rigging tool (section5).

- To generate an animation loading tool that allows the user to load animations from an animation file with a different skeleton structure to a rigged $3D$ model.

The contributions mentioned in the "Generation of an animation Rig" field has been published in [36] and [37]. Contributions made to the "Skinning" field has been sumbmited and accepted for publication in [38].

1.1 Thesis overview.

After this brief introduction we will discuss about automatic skeleton extraction and will adjust a predefined logic skeleton to a $3D$ model in Chapter 2. Chapter 2 starts with the state of the art in skeleton extraction, followed by all the algorithms and methods used to extract and adjust an extracted skeleton, ending with a results section.
In Chapter 3, mesh segmentation will be introduced; starting the chapter with a state of the art in segmentation and a discussion about how it is applied to computer animation, followed by the methods proposed to segment a polygon mesh, ending with its respective results section.
Chapter 4 will be focused in Skinning; starting with the state of the art in skinning methods, followed by a *LBS* skinning weight assign method

section, which uses one of our proposed segmentation algorithm as an input. Finally we will show an innovative deformation algorithm without the candy wrapper artifact based in the *LBS* algorithm and the segmentation of a $3D$ mesh character, ending this section with a discussion about the volume preservation feature of the proposed method.
The implementation of the tools to rig a $3D$ model will be discussed in Chapter 5. The implementation of the algorithms developed in this thesis will work as a set of commands developed for *Autodesk Maya*, loaded through their plug-in interface, followed by the explanation about an animation loader tool that can be used to load animations in the *BVH* file format with a different structure than the input rigged model.
The conclusion of this thesis will be in Chapter 6, where the results and limitations of each of the algorithms developed in this thesis are shown; followed by a comparative between some of the main algorithms and methods of each area, and concluding with ideas about improvements and future work.

Chapter 2

Automatic skeleton extraction.

Although the skeletization of 3D models is not directly related with the computer animation field; it's a useful tool to make a geometric skeleton. This geometric skeleton can be used as a basis to adjust a template logic skeleton; witch is one of the main elements in skeleton driven animation. As it is defined in the work published in [5], a skeleton (also known as curve-skeleton) is defined as the locus of centers of maximal inscribed (open) balls (or disks in 2D). More formally, let $X \subset R^3$ be a $3D$ shape. An (open) ball of radius r centered at $x \in X$ is defined as $Sr(x) = y \in R^3, d(x, y) < r$; where $d(x, y)$ is the distance between two points x and y in R^3. A ball $Sr(x) \subset X$ is maximal if it is not completely included in any other ball included in X. The skeleton is then the set of centers of all maximal balls included in X.
In [5] a set of desirable curve-skeleton properties are defined such as:

- **Homotopic** (topology preserving)
- **Invariant under isometric transformations**
- **Reconstruction**(it refers to the ability to recover the original object from the curve-skeleton)
- **Thin**(curve-skeletons should be one-dimensional: that is one voxel thick at most in all directions, except at connectivity voxels)
- **Centered**
- **Reliable**(it refers to the property of the curve-skeleton where every point on the objects surface is visible from at least one curve-skeleton)

- **Junction Detection and Component-wise Differentiation**(the curve-skeleton should be able to distinguish the different components from the original object, reflecting its part/component structure)

- **Connected**

- **Robust**(a desirable property of the curve-skeleton is to exhibit a weak sensitivity to noise on the objects surface)

- **Smooth**

- **Hierarchy**(a hierarchical approach is useful because it can generate a set of curve-skeletons of different complexities that can be used in different applications), and related to the algorithm used to compute the curve-skeleton.

Some applications need one or both of the following properties: **efficient**(if the algorithm used to compute the skeleton needs to feed a real time application) and **can handle point sets**(where the connectivity is not specified and there is no inside/outside information). Not all properties are useful for some applications; furthermore some of this properties may be conflicting each other because of their nature.
The process of extracting the skeleton from a $3D$ model is called skeletonization. The skeletonization methods can be classified in the next categories:

- **Thinning and boundary propagation:** The thinning methods produce curve skeletons by removing voxels from the surface of a solid voxelized $3D$ model iteratively; until the desired thickness or thinness is obtained. All the algorithms of thinning operate in a discrete space (a voxel set) and are based in the concept of *simple point*. A *simple point* is an object that can be removed without changing the topology of the object. The *simple points* have the property of being locally characterized; this is important because we can know if a voxel is a *simple point* by inspecting its neighborhood. If a *simple point* is removed; the algorithm has to take care of not removing any *end point*(which are also *simple points*) in an excessive way, because it will produce a shrinkage of the curve-skeleton branches. To avoid this condition, additional conditions must be added to the thinning algorithm to maintain the geometric properties of resultant curve-skeleton. The thinning algorithms can be classified into two categories based in the way of computing the *simple points* of a voxel set.

- **Sequential thinning:** Within this category we can find two kinds of algorithms: *subfield sequential thinning* and *directional sequential thinning*. The *subfield sequential thinning* is a kind of method that divides the space into several subfields, and at each sub-iteration a set of voxels belonging to a subfield are considered for deletion. *Directional sequential thinning* algorithms works similar to *subfield sequential thinning*; but instead of grouping the voxels into a subset by using some sort of distance function, the deletions of voxels are made by tagging them as candidates to removing some of the surface voxels when the algorithm is working in a sub-iteration (a specific direction: up, down, front, rear, left, right); at the end of the iteration depending of the properties of the candidates and the set of rules of the algorithm, some of the candidate voxels are going to be removed from the surface of the object [33].
- **Fully parallel:** This algorithms take into account all the surface voxels for their deletion in a single iteration. To maintain the object's topology the neighborhood must be inspected; to decide if a voxel is deletable, its neighborhood must be greater than the 26 neighbors. Some algorithms use templates to produce a curve-skeleton; others uses a sophisticated set of rules to delete surface voxels in the first stage, and continue until a curve-skeleton with a single voxel size is produced ([26][32]).

- **Distance Field:** In this skeletonization category, the distance field is defined for each interior point P of a three dimensional object O. A great number of distance functions can be used, such as Euclidean distance [45], or any other desired distance function. In [3], a distance field algorithm is used as a base to create a medial surface by choosing the C^1 discontinuities in the distance field; from the medial surface a process based on the distance to the components of the medial surface to the input mesh surface vertices is applied to create a set of spheres. Finally the centers of the spheres will be the vertices of a graph. This graph is an unrefined approximation of the input mesh's shape for their skeleton embedding.

- Geometric Methods:This methods are applied to objects represented by polygonal meshes or to points sets. Within this category we can find: **Voronoi diagrams**. A *Voronoi diagram* is generated by the vertex of a $3D$ polygonal object or directly from a set of disorganized points. The edges and faces of the *voronoi diagram* can be used to

extract an approximation of the medial surface; this medial surface can be reduced (or pruned) to a $1D$ structure that will be used as the skeleton.
Cores and M-reps, this methods are also based in medial surfaces: A *core* is a set of points in space which coordinates are position, radius, and associated orientations. *M-reps* are a generalization of *cores*; the *M-reps* model the medial surface using a set of connected atoms. **Reeb graphs**: this descriptors has its roots in the Morse theory; they are $1D$ structures that encode the topology and geometry of the original shape [43]. After being computed, the *Reeb graph* can be embedded by mapping the edges of the *Reeb graph* $3D$ points that will define a curve-skeleton of an object.
An alternative method can be found in [34], where a skeleton is extracted from a $3D$ mesh by taking two silhouettes ($2D$ projections of visible vertices from some visualization angles), using the silhouette with the highest number of visible vertices as the primary silohuette. Then the problem is treated as a triangulation problem in $2D$ where some central points are computed in $2D$ and finally, using the second silhouette, the deep component is computed.
In [4] a contact graph is built for each mesh input, then from each mesh a point cloud is created and if any point of that point cloud has a defined distance with another point of a different input mesh, a link is generated between the two points, creating a first skeleton approximation that is refined by a graph clustering algorithm.

- **General-field functions:** Is one of the ways used to extract curve-skeletons. *General-field functions* functions are not based in a distance function or transformation, but functions that mimic potential field, electrostatic field or a sort of repulsive force where the potential of a point internal to an object is determined by the sum of all the potential generated from the object's surface. In the particular discrete case, the voxels in an object's frontier are considered charged points that generate a potential field, then a skeleton is extracted by linking the detected filed local extremes.

2.1 Skeleton extraction.

To adjust a logic skeleton to an arbitrary $3D$ model we need to extract it from the $3D$ model. Our main objective in this chapter is to obtain a logic skeleton from an extracted one. To achieve this objective, we needed

a simple and effective method of skeletonization; from the diversity of algorithms exposed in the previous section, we believe that the ones based on a discretized space (a set of voxels) are good and efficient enough to our particular purpose (see [5] for a more general approach) of the universe of skeletonization methods. Therefore we had chosen a sequential thinning method as skeleton extraction algorithm.
Thinning algorithms are based on removing surface voxels from a voxel set that represents a solid object. The particular algorithm we are using was published in [33]; it removes voxels from the surface until it reaches the point where the voxels that remain will retain the original shape and topology of the input voxel set. Algorithm [33] was chosen because of its simplicity and low computation cost; more sophisticated and complex thinning algorithms exist but their main target is to be applied in parallel architectures. Although we can use parallel algorithms like [26] and [32]; algorithm [33] fulfills our main objective which is to adjust an animation skeleton to a skeleton extracted from a model's mesh.

Mesh voxelization: Surface and interior voxels.

We decided to implement our own voxelization method in order to have a complete pipeline inside of our final Maya plug-in. Sometimes the skeleton can be provided by the animation team, but in the case of starting from only a triangulated mesh we need this kind of tool to continue our rigging process. Moreover, the final geometric skeleton will be associated with a logical one (for instance taken from a motion capture data) and we need to achieve enough precision in this step. As it will be explained later, we consider in our pipeline the possibility of user interaction in case the automatic results must be improved.

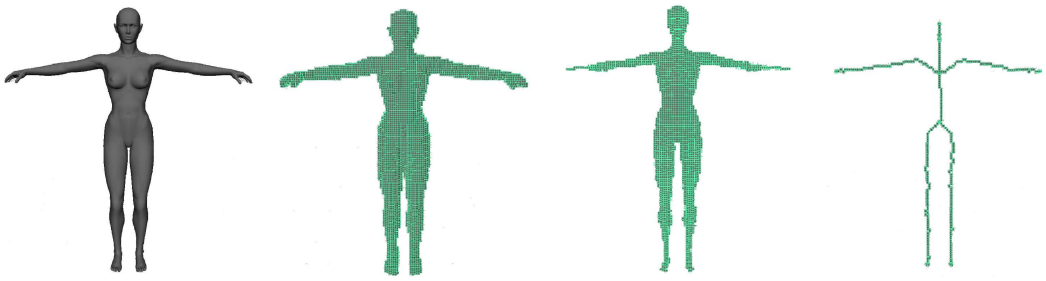
Because we decided to apply a thinning approach to compute the skeleton, the first part of the algorithm consists on building a voxelized model from the original mesh. This goal can be achieved with high performance using GPU approaches (see [5] and [31]) but in our case we implemented a non-optimized version based on two steps:

1. First we voxelize the original triangles. This is how we obtain the surface voxels of our model.

2. Then we fill the voxelized surface by using a 3D flood algorithm, obtaining the interior voxels of the model.

This is the information needed to apply our thinning algorithm approach described in [33]. As we pointed out, there are other approaches [5] not

based on thinning, like geometric approaches to compute the skeleton or general potential-field methods. Although sometimes they can be more efficient, we have chosen the thinning one because it's very intuitive and easy to code in our plug-in. Of course, this can be changed in the case of a greater speed being needed.



(a) *Original polygon mesh.* (b) *Voxelized surface.* (c) *Set of filling voxels, without fusion in the legs.* (d) *Thinned solid.*

Figure 2.1: *Transformation of a polygon mesh into a skeleton.*

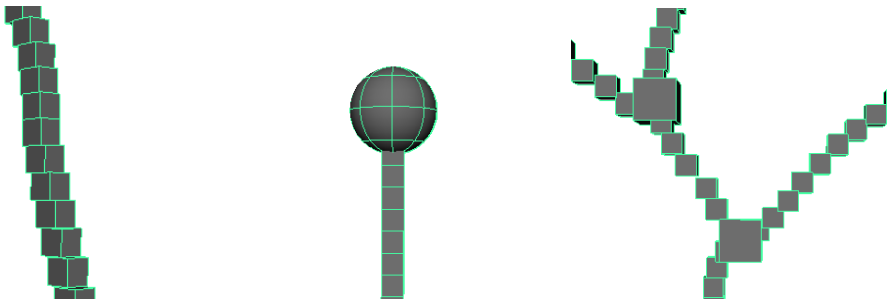
Sequential thinning algorithm.

The detail algorithm [33] is based on the concept of removing surface voxels in a set of voxels in a voxelized solid until it has reached a state where no more voxels can be removed without affecting the original shape and topology of the resultant solid; the remain set of voxels can be considered as a *geometric skeleton*. The thinning algorithm is applied iteratively in six directions (one for each face of a voxel): *UP, DOWN, LEFT, RIGHT, FRONT, REAR*. These stages of the algorithm are called *sub iterations*. Each time that a sub iteration is applied to the solid, a set of n surface voxels are removed depending on a set of rules. The algorithm stops if the number n of deleted voxels is equal to zero. Each sub iteration is basically the same process, but applied in different directions, specifically the set of surface voxels were the sub iteration is applied. The directions on the sub iteration process are defined by the neighborhood on the surface voxels. If the direction is *UP* that means that the process is going to be applied to all the surface voxels whit a null voxel in its upper face.
In the development of this thesis we have programmed algorithm [33] with some modifications and optimizations.

2.2 Voxel classification.

The extracted skeleton obtained with the algorithm depicted in section 2.1 gives us more information about the voxels that are part of the skeleton than only about their position. This extra information is based on the number of neighbors that have a voxel in a skeleton. Therefore we can infer which part of the skeleton has correspondence with a part of a human-like skeleton. A voxel from the extracted skeleton can be classified by the number of neighbors within its 26-adjacency in the next categories:

- Flow nodes: This are voxels with two elements in its neighborhood. The flow voxels are named like that because they are part of the segments that represents the limbs in a skeleton.
- End nodes: Voxels with only one neighbor voxel in their neighborhood. They usually represent the end of the limbs (arms, legs, fingers, etc.) or the head.
- Connection nodes: Voxels with more than two elements in their neighborhood. These voxels usually represent a solid-rigid part of the model; like the hips or the chest.



(a) *Flow node.* (b) *End node.* (c) *Connection node.*

Figure 2.2: *Voxels classified by their neighborhood.*

To classify the voxels in the extracted skeleton we have two options: To check all the voxels in the space to classify the ones who are part of the skeleton, or use an algorithm that classifies the extracted skeleton's voxels by traversing all the voxels within it. We have chosen the second option by creating a simple algorithm that traverses the voxels of the extracted skeleton to fulfill our needs.

2.3 Creation and refinement of a geometric skeleton.

The traversal algorithm is simple, it allow us to transform the extracted skeleton into a data structure to extract useful information such as: the number of segments that are connected, the length between two elements in the data structure, or the length of a segment of consecutive flow nodes.

Algorithm 2.3.1 Voxel traversal algorithm.

Require: v ▷ A random voxel in the extracted skeleton.
Require: *inter* ▷ A stack to store voxels in the traversal.
1: $pushStack(inter, v)$
2: **while** $StackLenght(inter) > 0$ **do**
3: $neighNum \leftarrow GetNumNeighbors(v)$
4: **if** $neighNum > 2$ **then**
5: $RegInternaNode(v)$
6: $pushStack(inter, v)$
7: **else if** $neighNum = 1$ **then**
8: $RegEndNode(v)$
9: $v \leftarrow popStack(inter)$
10: **else**
11: $RegFlowNode(v)$
12: **end if**
13: $v \leftarrow getFirstNeighborLeft(v)$
14: **end while**

function $GetNumNeighbors(v)$
Returns the number of neighbors of the voxel v.
end function

function $getFirstNeighborLeft(v)$
Returns the first unregistered neighbor of the voxel v
end function

Geometric skeleton data structure.

We have chosen a *n-ary* tree data structure to map an extracted skeleton into a data structure that we call: *geometric skeleton*. We have two main advantages form representing a geometric skeleton as a tree data structure:

1. ***Fast and easy traversal over all the skeleton:*** When the thinning procedure has been applied to the model; we define a node for each remaining voxel. All the operations (position change, neighbor-

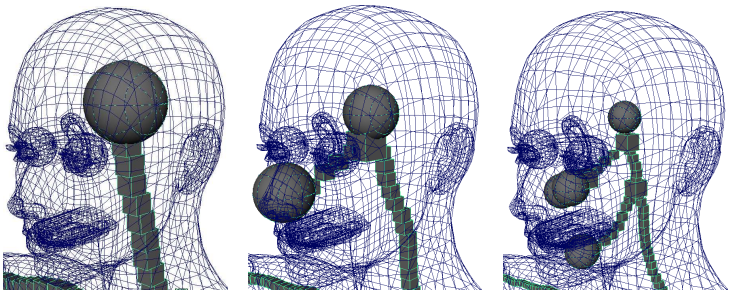
hood and classification of the nodes) done over the voxelized space are applied and stored in a data structure.

2. ***Allows us to perform operations over nodes:*** Modify or delete a node or an entire set of nodes (loops).

Geometric Skeleton refinement.

Once the geometric skeleton is created, we apply a post-process to refine it. This post-process will have the following steps:

1. ***Deleting of loops and redundant nodes:*** The result of the thinning process over a voxelized model is a set of voxels that represents a skeleton. Usually, this set has voxels which are noisy or redundant nodes (voxels which cannot be removed because of their topology condition [33] on the thinning stage). We must have in mind that the size of the voxel in our space can change the number of details and noise in the geometric skeleton. If the voxel size is small, the thinning algorithm tends to introduce more voxels as end nodes, this will generate more branches in the geometric skeleton (fig. 2.3).



(a) Voxel size at 1% of height. (b) Voxel size at 0.65% of height. (c) Voxel size at 0.4% of height.

Figure 2.3: *Extracted skeleton at different voxel sizes.*

2. ***Root node adjusting:*** Because we use a random voxel as starting point in the creation of a geometric skeleton, the root node must be adjusted. Only connection nodes can be root nodes; the main reason is that practically in all animation formats the hips are taken as the center of mass for translations and rotations; the hips can be considered as a solid-rigid. If the root node in the geometric skeleton is not a connection node, the nearest connection node will be assigned as

the root, and the geometric skeleton tree will be balanced for the new root node. The assigned connection node is the first approximation to the model's hip; the appropriate assignment will be done in a posterior step.

3. ***Skeleton smoothing:*** A smoothing step is mandatory because in a voxelized space, change of position between nodes of the skeleton in the same neighborhood are produced in the edges of the voxel. This will lead to undesirable artifacts if this data is used to calculate direction changes between two voxels. By changing the position of the voxels that share an edge as their contact surface to a position where they share part of their faces, a smooth transition is granted. We use a window based method as our smoothing process.

2.4 Logic skeleton adjustment.

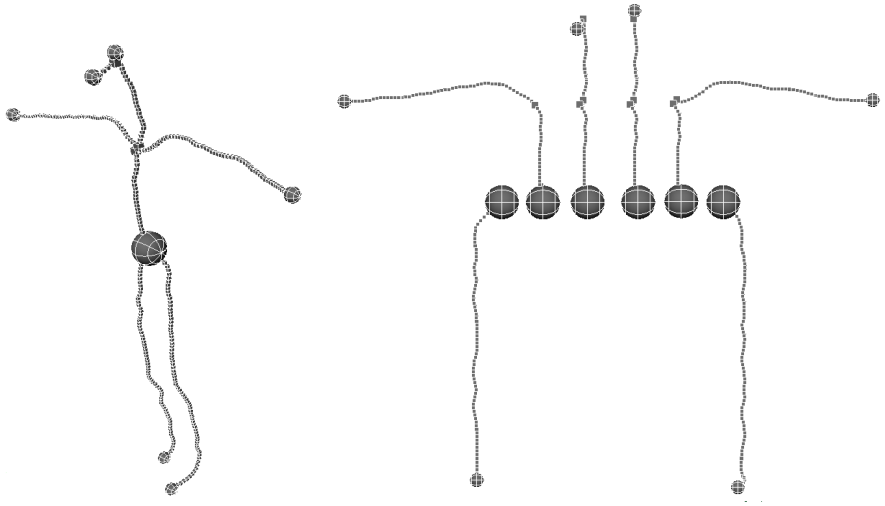
Segments are the core elements in the adjustment of a logic skeleton (rig), to a geometric skeleton. We define a segment as:
Segment: *A set of nodes traversing the skeleton from a connection node to an end node.(fig.2.4. b.).*
Using our definition of segment, a skeleton (geometric or logic) can be defined as:

Skeleton: *A set of segments with the same connection node as starting point (fig.2.4. a).*

In full body animation, only five end nodes are needed (head, hands and feet) [3], furthermore the great majority of full body motion capture data is produced with five end nodes [1]. Therefore we have restricted our method to logic skeletons with five end nodes.

2.5 Node selection and root assignment.

The main problem of adjusting a logic skeleton to a geometric skeleton is finding the correspondence between their body segments (head, hands, and feet). Logic skeleton's limbs are specified by a tag which can be obtained from a file, a user interface, or if the model was in a specific pose it can be tagged automatically by the positions of its segments in the space.



(a) Extracted skeleton. (b) Isolated segments.

Figure 2.4: *Extracted skeletons and their segments, the third segment (root to nose) will be deleted.*

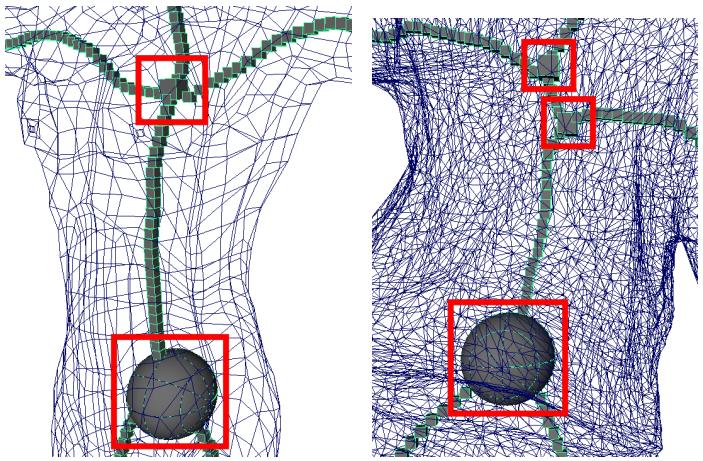
End node selection.

Geometric skeleton's limbs are not specified or tagged, mainly because the input 3D models can be in an arbitrary pose; therefore, there is no simple method capable of automatically tagging the limbs of a 3D model; moreover, there are models with human like forms but with an extra limb (for instance the tail of an armadillo model). Limbs detection is a difficult and challenging task that is out of the scope of this thesis. To solve this problem we have implemented an interface that allows the user to select which are the end nodes that correspond to their appropriate limbs.
In our user interface, the end nodes are marked with a sphere and the flow nodes are represented by small cubes(big cubes represent connection nodes). The user must decide which end node corresponds to its logic limb by selecting the appropriate sphere (fig. 2.3 b and c).

Root assignment.

Once the limbs are assigned; we delete all the nodes that are not part of an assigned segment.(fig.2.4 b.).
When segments are assigned, the number of connection nodes will decrease and only connection nodes that represent non-articulated parts of

the model(hips and chest) will be preserved.
It is customary to set the hip as the root node; in our case, the hip will be one of the connection nodes but depending on the number of connection nodes the next situations can arise:



(a) Skeleton with two connection nodes.

(b) Skeleton with three connection nodes.

Figure 2.5: *Root assignment cases.*

- ***Two connection nodes:***In this case, the difference between the chest and the hip is caused because the chest will have three segments without connection nodes(the hands and the head, fig. 2.5 a.), and the hip will have two (the feet). To apply this rule we are going to build two sets of segments (one per connection node); each set of segments will have its starting node in one of the connection nodes. Finally we will assign the set with the least number of segments as the hip (root) of our skeleton.

- ***Three connection nodes:*** In this case we calculate the addition of the Euclidean distances between flow nodes from one connection node to another. The two nearer connection nodes will represent the chest and the other one the hip. Therefore, to find the hip we create three set of segments: one per connection node. For each set we select the segment with the minimum number of flow nodes between the segment's starting node and its nearest connection node, then from these three segments we choose the one with the maximum number of

flow nodes. The starting node of the selected segment will be assigned as the hip (root) of the skeleton.

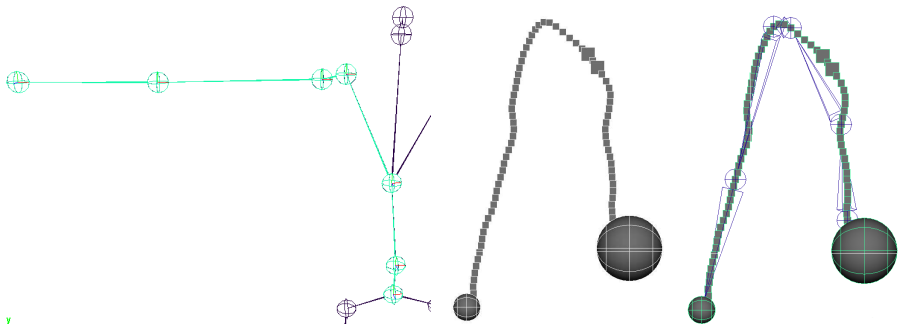
Skeleton adjustment.

A logic skeleton can also be viewed as a set of segments. If we have followed the previous steps correctly; we must have the same number of segments in the geometric and logic skeletons but in the logic skeleton we will have additional tagged nodes (elbow, neck, ankle...) that are not tagged in the geometrical one. Adjusting a logic skeleton to a geometric one is reduced to finding the correspondence between logic tagged nodes and geometric untagged nodes.

Scaling segments.

As it is mentioned in the section 2.5, our skeletons will be represented by a set of five segments. Since a segment in the logic skeleton has its equivalent in the geometric skeleton, we can define a normalized distance in our skeleton's segments; being *zero* the starting node position and *one* the end node position. With this distance we can find the position of the logic skeleton's tagged nodes and map them to the remaining geometric skeleton's untagged nodes (as an example, the segment in the geometric skeleton that represents an arm will have the elbow, shoulder and other untagged nodes that will be tagged in the logic one).
The distance of the logic skeleton segments is defined as the sum of the distance between two neighbor nodes (joints) in a segment from the root node to the end node. We have defined the distance of the geometrical skeleton segment as the sum of the distances between the center of two neighbors nodes (voxels) in a segment from the root node to an end node. Since a geometric skeleton can have a different pose than that of the logic skeleton, only the distance between nodes of the geometric skeleton will be used to map from the logic skeleton's internal nodes to their equivalent in a geometric skeleton. Suppose we have a segment SL_i in the logic skeleton with n internal nodes and their equivalent segment in the geometric skeleton SG_j with m internal nodes and $m \geq n$; then, the mapping process will start from the initial nodes $vl_0 \in SL_i$ and $vg_0 \in SG_j$ (that are equivalents) and we will traverse each internal node $vl_k : 1 \leq k \leq n$ in the logic skeleton and find their equivalent within the geometric skeleton nodes set $vg_l : 1 \leq l \leq m$ that have approximately the same normalized distance of vl_k for being tagged. Basically, adjusting a logic skeleton to a geometric skeleton is finding a partition of the node graph formed by the logic skeleton segment,

and map their internal nodes to the set of untagged nodes that are available within the geometric skeleton segment. The union of all this mapped nodes (with its hierarchy implicit) will be the adjusted skeleton.



(a) Logic skeleton segment. (b) Geo. skel. segment. (c) Logic segment adjusted to a geo. skel. segment.

Figure 2.6: *Segment adjustment.*

2.6 Results.

In fig.2.6 we show the results obtained by applying our method to arbitrary models in different positions, the voxelization and skeletization time will depend on the model's pose and its number of triangles. The chosen voxel size is 0.65% of the model's height with processing times in the range of 2 and 3 seconds. The geometric skeleton creation, and the logic segment adjustment processing time will be increased if more connection nodes and segments are obtained. Our times are in the range of 2 to 3 seconds for models with a density of 20K and 28K triangles (a more detailed discussion about results can be found in 6.1).

The skin attachment of the skeleton has been done through Maya's mesh binding that generates automaticlly the set of weights for the LBS(the default deformation method used in Maya) as can be seen in figure 2.7 last column.

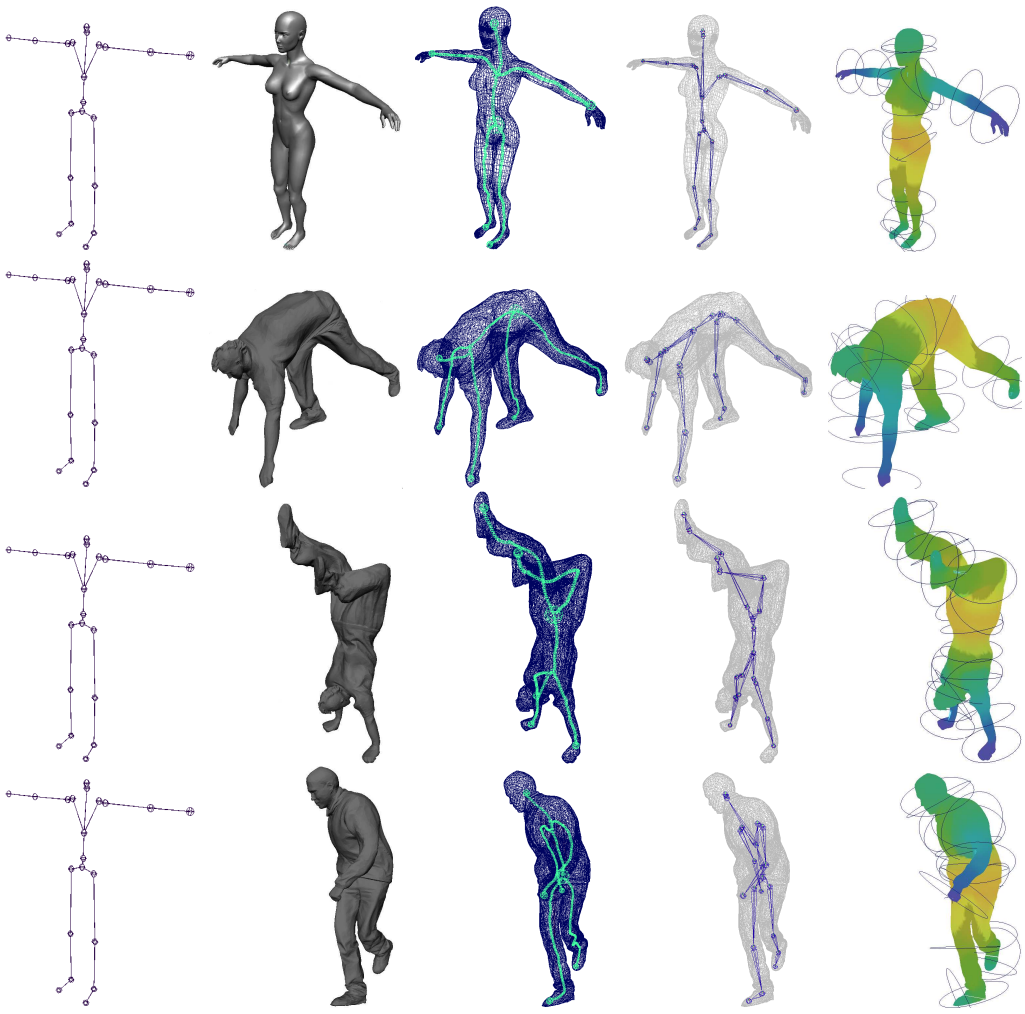


Figure 2.7: *Columns:Skeleton from an animation file, arbitrary model, geometric skeleton, adjusted logic skeleton, binded model.*

Chapter 3

Mesh Segmentation.

The main objective of the 3D mesh segmentation is to make a partition of an object into smaller objects (or patches if we take the segmented object as a surface), with a specific purpose or application in mind. In computer graphics, mesh segmentation has different applications like: texture mapping, morphing, compression and animation. The main categories for segmentation algorithms according to [40] are *part-type* segmentation and *surface type* segmentation. *Part-type* segmentation is oriented to partitioning the object into semantic components; *surface type* uses geometric properties of the mesh to create surface patches.
Another common application for mesh segmentation is skeleton extraction: an input mesh is taken and partitioned in segments that will represent a region that belongs to a skeleton bone. An example can be found in [6], where a segmentation method is described. The method takes a set of meshes that represent an animated mesh sequence through time as an input, then the input mesh is segmented in patches that undergo approximately the same rigid transformation over time. A curvature-based segmentation method is used to decompose the model into l surface patches (*part-type* segmentation); then a skeleton is estimated by finding the mesh's kinematic topology (where parts of the input mesh body are adjacent) using the patches of the segmentation stage. In [14] an indirect segmentation is made: first an estimation of the mesh's bones is made using a set of related meshes (that represents the desired set of deformation); the estimation is made using a main shift clustering algorithm, that computes an estimated number of bones in a density function gradient, the estimation is made over a vertex in the target mesh in a period of time; then the mesh vertices are mapped to a set of stationary points that will be used to estimating a set of bones statistically. The described process will be used in a modified version of the **LBS** algorithm. Our approach is different because instead of extracting a

skeleton from a previously segmented mesh, we segment a mesh using an existing skeleton. The purpose of segmenting a mesh is to create what is known as a *rigid skinning*, using it as the initial value for the weights of the **LBS** skinning algorithm. One of the algorithms shown in this thesis is similar in essence to the one showed in [39]: Our segmentation method based in voxelization. The mentioned algorithms share some characteristics: both use a defined skeleton to perform the mesh segmentation, use voxelization of the input mesh, and are used as an initial guess to compute skinning weights. But our main algorithm is based on *part-type* partition segmentation, applied to a rigged mesh instead of a single input mesh.

3.1 Segmentation of a polygon mesh.

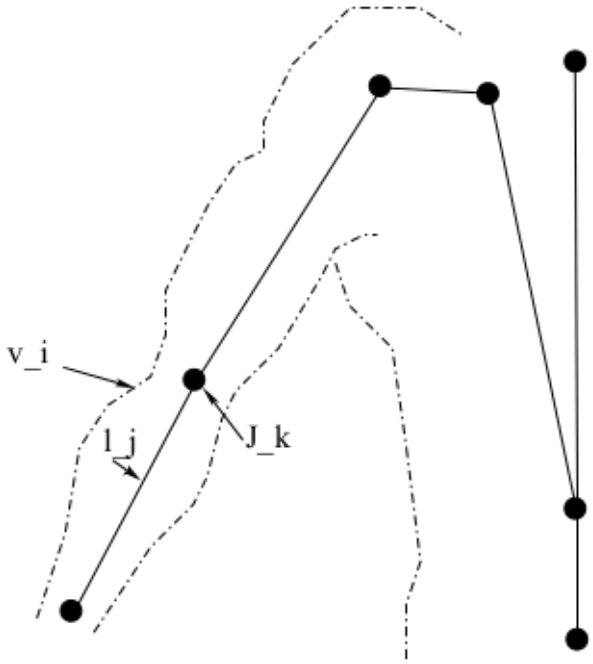


Figure 3.1: *A point of a Mesh.*

Our first segmentation algorithm is based in measuring the distance between the vertex and the elements of the rigged skeleton (joints and links). For each vertex v_i we compute two distances: the distance Jd to the closest joint J_k in the skeleton, and the distance ld to the closest link if v_i is on the region delimited by each link. When we say that a vertex is on a link's region we mean that a vertex v_i is inside the cylinder of infinite

radius made by a link l_j.
The first step of the segmentation algorithm is made into a list LJd_i of distances Jd. The list is sorted using Euclidean distances as a sorting parameter; therefore the minimum distance Jd will be the first element on the list.

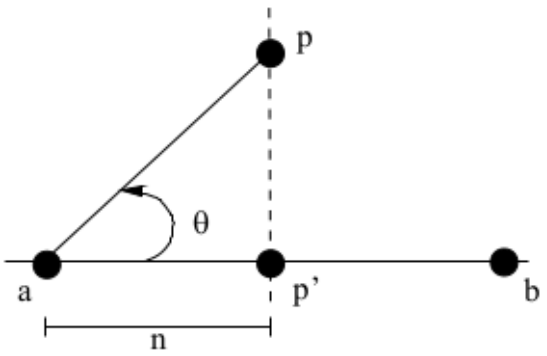


Figure 3.2: *Projection of a point p on the link (vector) formed by* $(b - a)$ *line segment.*

The second step is to create the list Lld_i that stores the distance of a vertex v_i to a link l_j, but the list Lld_i will be filled only with the region's links were the vertex v_i is inside. To know if a vertex v_i is on the region of a link l_j we use a δ function introduced in [44]. The δ function is applied to a vertex v_i and the elements of a link l_j (as is showed in the fig. 3.2). Basically it is a formula that takes a vertex v_i, the joints j_a and j_b (the joints that form the link) as input parameters and gives us an output. The output value will be positive if the vertex is on the link's region, with a value between 0 and 1; otherwise the vertex is out of the link's region.

The δ function

The δ function is defied as:

$$\delta = \frac{(p - a) \cdot (b - a)}{\|b - a\|^2} \tag{3.1}$$

In a shell, the δ function is the projection of the point p over the line parallel to the vector $(b - a)$ divided by $\|b - a\|$; δ is a function that allows us to know if the projection is outside or inside a vector. If the projected points

p' of p are inside $\vec{ba}$, the δ value will be between 0 and 1; if its outside but p' is after b, δ will be greater than 1; and if p' is before a, δ value will be negative, as can be seen in the next equations:

$$\delta = \frac{(p-a)\cdot(b-a)}{\|b-a\|^2} = \frac{\|p-a\|\|b-a\|\cos\theta}{\|b-a\|^2} = \frac{\|p-a\|\cos\theta}{\|b-a\|} \tag{3.2}$$

A relation involving the triangle formed by a,p',p and the vectors $(p'-a)$ and $(a-b)$ can be written:

$$\frac{\|p-a\|}{\|b-a\|} = \frac{\|p-a\|}{\|p'-a\|n} = \frac{1}{(\cos\theta)n} \tag{3.3}$$

substituting 3.3 in 3.2, δ is reduced to:

$$\delta = \frac{1}{n} = \begin{cases} \delta < 0 & \textbf{if} \quad n < 0 \\ 0 < \delta < 1 & \textbf{if} \quad n > 1 \\ \delta > 1 & \textbf{if} \quad 0 < n < 1 \end{cases} \tag{3.4}$$

With n being a scale factor of $(p'-a)$ over $(b-a)$, and δ being the reciprocal of n.

Distance to link.

If v_i is on the link l_j's region; the distance of v_i to l_j will be computed as:

$$l = \|(p-a) - \delta(b-a)\| \tag{3.5}$$

That is equivalent to the classic point-to-line length formula, because of the equivalences:

$$l = \left\| \frac{(p-a)\times(b-a)}{\|b-a\|} \right\|$$

$$= \frac{\|p-a\|\|b-a\|\sin\theta}{\|b-a\|}$$

$$= \|p-a\|\sin\theta$$

and

$$\sin\theta = \frac{\|p-p'\|}{\|p-a\|}$$

$$\cos\theta = \frac{\|p'-a\|}{\|p-a\|}$$

or

$$\|p - p'\| = \|p - a\| \sin\theta$$

$$\|p' - a\| = \|p - a\| \cos\theta$$

if

$$(p' - a) = \delta(b - a)$$

then

$$\begin{aligned}(p - p') &= p - (p' - a + a) \\ &= (p - a) - (p' - a) \\ &= (p - a) - \delta(b - a)\end{aligned}$$

finally

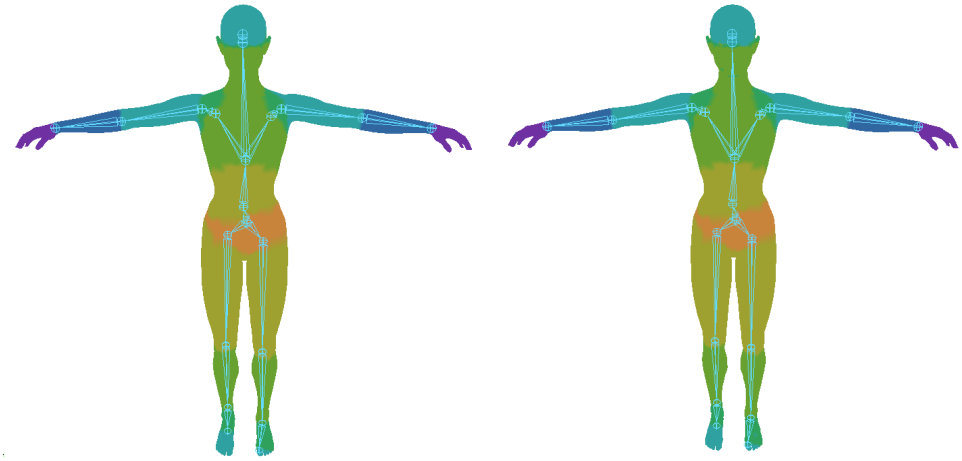
$$\begin{aligned}\|(p - a) - \delta(b - a)\| &= \|p - p'\| \\ &= \|p - a\| \sin\theta \\ &= l\end{aligned}$$

Joint distance and Link distance comparative.

Once the lists LJd_i and Lld_i are computed and sorted, we start by checking if the list Lld_i has elements (there is a possibility that the vertex v_i does not belong to a link region). If that is the case, we compare the minimum distances of the lists Lld_i and LJd_i, and the vertex v_i will be assigned to the closest segment defined in any of these two lists.
When the list Lld_i is empty (as in the case of vertices that forms the hand or the top of the head); v_i will be assigned to the first element of the sorted list LJd_i.
Finally we can have false assignments when a vertex is associated to a wrong link, because it belongs to a different part of the model. This can be detected when the line between the vertex and the corresponding link passes through a region that is external to the model. To detect this situation the best that can be done in an fast way, is using ray-triangle intersections [35]. In our case, because we have already build a voxelized representation of our model, and also for including other models not necessarily defined as a triangular mesh; we decided to use the voxelized space to determine when you pass through an external voxel, and therefore, discard this link. We have to admit that this is not an optimal solution and of course, it is not the one we will choose if the voxelized model was not available.

3.2 Segmentation algorithm.

We present a couple of pseudo-codes for the segmentation algorithm: the original (algorithm 3.2.1) and the voxelized version (algorithm 3.2.2.



(a) Segmentation algorithm. (b) Voxelized segmentation algorithm.

Figure 3.3: *Comparative between versions of the segmentation algorithm, mesh segmentation colors per link.*

The version including the voxelization is almost the same, but we use four lists instead of two to make the comparatives between distances (link distance, joint distance).
List one (dL_1) and list three (dL_3) will be used with the voxelizated mesh; dL_1 will have the list of joints that had a line without null voxels. If a line had a null voxel, is stored in dL_3 instead. dL_2 and dL_4 are used in the same way that dL_1 and dL_3 respectively, but are sorted by the distance between the links and a vertex. Lists dL_3 and dL_4 are in fact a precautionary measure if none of the two first list had elements. They will be filled and sorted using the distance from the joints and links to the vertex if the line between a vertex and joint, or a vertex and a link is outside the mesh. In algorithm 3.2.2 a post process is needed because this algorithm relies heavily in the voxelization of a Mesh. The result is dependent of the density of the voxelizated space (the voxel size). In some cases, a vertex is not assigned properly because of the voxelization density. This improper assignation can be solved by incrementing the density of the voxelizated mesh at expense of the computation time. But there are some cases where the mesh has not convex vertices. If that is the situation, the result will be the same

Algorithm 3.2.1 Segmentation algorithm.

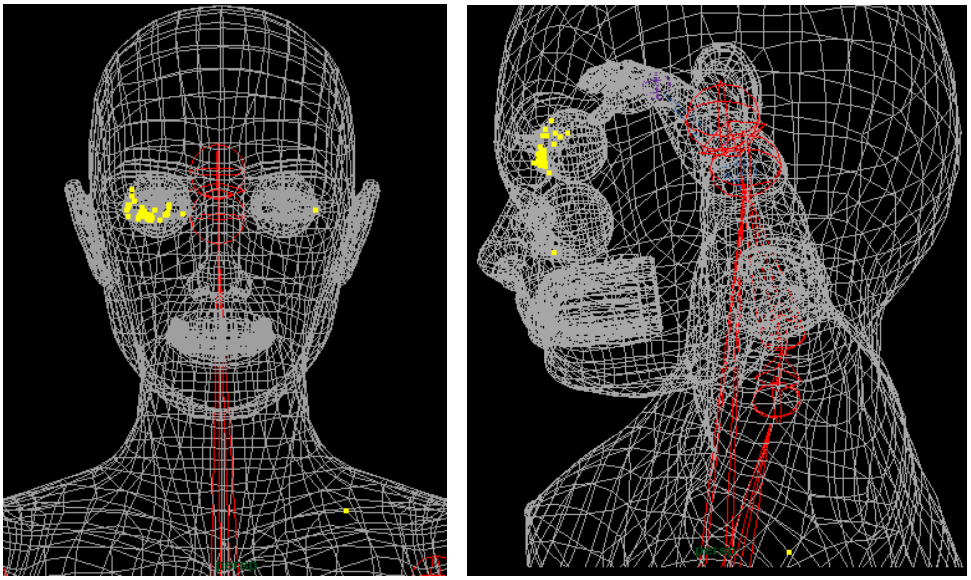
Require: JL ▷ List with the joints data.
Require: LkL ▷ List with the links data.

1: **for** $x = 0 \rightarrow nVtx$ **do**
2: $Vtx \leftarrow GetVertexData(x)$
3: $point \leftarrow VtxWCoord(Vtx)$
4: **for** $y = 0 \rightarrow nJoints$ **do**
5: $jntpnt \leftarrow JntWCoord(y)$
6: $jd \leftarrow euDst(point, jntpnt)$
7: $InsSortedList(dL_1, y, dJ)$
8: **end for**
9: **for** $y = 0 \rightarrow nLinks$ **do**
10: $lnkElem \leftarrow LnkData(y)$
11: $sf \leftarrow delta(Vtx, lnkElem)$
12: **if** $sf > 0$ & $sf < 1$ **then**
13: $ld \leftarrow euDst(point, lnkdata)$
14: $InsSortedList(dL_2, y, dLk)$
15: **end if**
16: **end for**
17: $JdlElem \leftarrow GetListFstElem(dL_1)$
18: $LnkdlElem \leftarrow GetListFstElem(dL_2)$
19: $elemFlg \leftarrow isValidElem(LnkdlElem)$
20: **if** $elemFlg$ **then**
21: $dLk \leftarrow GetLnkElmdst(LnkdlElem)$
22: $dJ \leftarrow GetJntElmdst(JdlElem)$
23: **if** $dLk < dJ$ **then**
24: $SetSeg(Vtx, LnkdlElem)$
25: **else**
26: $SetSeg(Vtx, JdlElem)$
27: **end if**
28: **else**
29: $SetSeg(Vtx, JdlElem)$
30: **end if**
31: **end for**

function $InsSortedList(Lst, indx, data)$
Inserts in a double list the element from the list Lst with index $indx$; the list is sorted using the value of the $data$ parameter.
end function

function $delta(Vtx, lnkElem)$
Compute the value of the δ function depicted in 3.1 over the vertex data Vtx, and the link data $lnkElem$.
end function

function $SetSeg(Vtx, Elem)$
Assign the vertex data Vtx to link in $Elem$. If $Elem$ is a joint, it assigns Vtx to the link where $Elem$ is the point a in the delta function.
end function



(a) Incorrect assigned vertices front. (b) Incorrect assigned vertices side.

Figure 3.4: *Comparative between versions of the segmentation algorithm, yellow vertices with different segment assigned.*

independently of whether the voxelization density and the analyzed vertex will be assigned to the closest link or joint that had a line without null elements.

Watershed based segmentation.

Algorithms 3.2.1 and 3.2.2, give us goods results in specific cases but are limited as algorithms for general purposes; in specific, 3.2.1 is limited to rigged meshes in *T-Pose*, and 3.2.2 is only applicable to closed rigged-meshes, and its computation and memory consumption are high and in some cases inviable. To solve this particular situation we have designed an algorithm based on part-type segmentation. In our particular case the semantic parts of the part-type segmentation are already created: the underlying logic-skeleton. Therefore our algorithm has the purpose of detecting vertices that belong to each semantic part. In this case, that will be the link between a joint and its child. Although our algorithm works for non-closed and multiple meshes, we will explain the method with the assumption that we have a single closed mesh as input. In the same fashion, like algorithms 3.2.1 and 3.2.2, we map the skeleton to a tree data structure, which allows us to tra-

Algorithm 3.2.2 Voxelized segmentation algorithm Part 1.

Require: JL ▷ List with the joints data.
Require: LkL ▷ List with the links data.
Require: $VxSpc$ ▷ Solid voxelized version of the mesh.

for $x = 0 \rightarrow nVtx$ **do**
  $Vtx \leftarrow GetVertexData(x)$
  $point \leftarrow VtxWCoord(Vtx)$
  for $y = 0 \rightarrow nJoints$ **do**
    $jntpnt \leftarrow JntWCoord(y)$
    $lnFlg \leftarrow isInOutLine(VxSpc, jntpnt, point)$
    $dJ \leftarrow euDst(point, jntpnt)$
    if $lnFlg$ **then**
      $InsSortedList(dL_1, y, dJ)$
    else
      $InsSortedList(dL_3, y, dJ)$
    end if
  end for
  for $y = 0 \rightarrow nLinks$ **do**
    $lnkElem \leftarrow LnkData(y)$
    $sf \leftarrow delta(Vtx, lnkElem)$
    if $sf > 0$ & $sf < 1$ **then**
      $lnFlg \leftarrow isInOutLine(VxSpc, lnkdata, point)$
      $dLk \leftarrow euDst(point, lnkdata)$
      if $lnFlg$ **then**
        $InsSortedList(dL_2, y, dLk)$
      else
        $InsSortedList(dL_4, y, dLk)$
      end if
    end if
  end for
  $VxJdlElem \leftarrow GetListFstElem(dL_1)$
  $JdlElem \leftarrow GetListFstElem(dL_3)$
  $VxLkdlElem \leftarrow GetListFstElem(dL_2)$
  $LkdlElem \leftarrow GetListFstElem(dL_4)$
  $elemVxLFlg \leftarrow isValidElem(VxLkdlElem)$
  $elemVxFlg \leftarrow isValidElem(VxJdlElem)$
  if $elemVxLFlg$ **then**
    if $elemVxFlg$ **then**
      $dLk \leftarrow GetLnkElmdst(VxLkdlElem)$
      $dJ \leftarrow GetJntElmdst(VxJdlElem)$
      if $dLk < dJ$ **then**
        $SetSeg(Vtx, VxLkdlElem)$
      else
        $SetSeg(Vtx, VxJdlElem)$
      end if
    else
      $SetSeg(Vtx, VxLkdlElem)$
    end if
  else if $elemVxFlg$ **then**
    $SetSeg(Vtx, VxJdlElem)$
  else
    $elemLFlg \leftarrow isValidElem(LkdlElem)$
    $elemFlg \leftarrow isValidElem(JdlElem)$

Algorithm 3.2.3 Voxelized segmentation algorithm Part 2.

```
50:         if elemLFlg then
51:             if elemFlg then
52:                 dLk ← GetLnkElmdst(LkdlElem)
53:                 dJ ← GetJntElmdst(JdlElem)
54:                 if dLk < dJ then
55:                     SetSeg(Vtx, LkdlElem)
56:                 else
57:                     SetSeg(Vtx, JdlElem)
58:                 end if
59:             else
60:                 SetSeg(Vtx, LkdlElem)
61:             end if
62:         else
63:             SetSeg(Vtx, JdlElem)
64:         end if
65:     end if
66:     PostProcAssg(Vtx, 1)
67: end for
```

function $isInOutLine(VxSpc, lnkdata, point)$
Returns true if the line in the voxelizated space has only solid elements; false if the line has empty voxels (null value voxels).
end function

function $PostProcAssg(Vtx, hLevel)$
Check hierarchically the distances from the links of the skeleton to the vertex Vtx. hLevel will be the hierarchical grade of parents and children of the elements used in this process. If hLevel value is 1; the check will be on the first grade parent and children of the link assigned to Vtx. If one or more of the links traversed during this check are closer to the vertex Vtx than the assigned at that moment; then Vtx will be assigned to the closer link in the set.
end function

verse the skeleton by its hierarchy. Our mapping of the skeleton is not by joints, but instead we use a pair of joints: joint j_a and its child j_b (a segment that we will reference as s_j) to define a node in our tree data structure; that has end joints as special cases. Therefore, a skeleton will have m number of segments for a skeleton with n joints, with $m > n$ (m is larger than n due to the fact that the end nodes are counted as segments of their own) and will be related directly by their hierarchy depending on their position within the tree data structure. Our algorithm is composed by three stages:

1. Region growing. In this stage we assign to each vertex, a set of segments that can be the segment where it belongs.

2. Belonging test. For each candidate segment in a vertex defined in the previous step, we apply a set of rules to discriminate which is the most suitable segment to be assigned.

3. Region merge. If false positives exist, we merge them with one of their surrounding neighboring regions.

Region Grow method.

We start our algorithm using one of the root node related segments as initial growing region; as any region grow algorithm a seed is needed; in the case of the first segment we can use a vertex manually chosen, or we can use the closest vertex measured in Euler distances to the root node that belongs to the initial segment. We apply our test for each segment in the skeleton hierarchically, using the delta function combined with region-grown as tool to check if this vertex is candidate to being inside a segment for each vertex traversed.

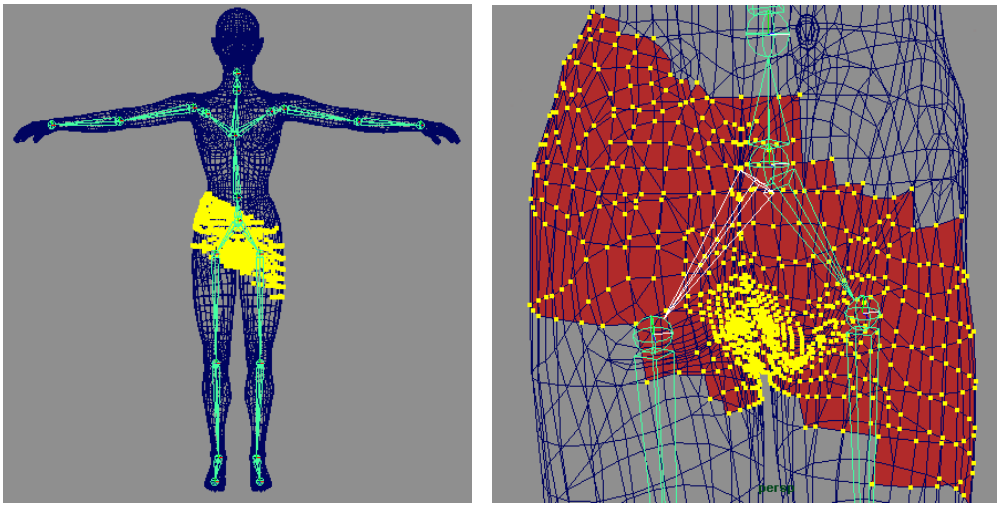
Region-grow needs a vertex as a seed to begin with. In our particular case, we used a vertex that had a delta value between 0 and 1 as seed (the vertex is in the influence space of the segment). If the value of delta is greater than 1, we compute the delta value output for the child segment (next segment in hierarchy). If its value is not in the child segment influence space; we mark it as candidate for being part of the current segment; therefore only vertices with delta value greater than 0 can be candidates if all the conditions are accomplished, and as consequence only a portion of the vertices of the mesh are checked per segment.

The region-growing method marks a vertex for an specific segment as a candidate; therefore when all the segments are computed, we will have a list of segments for each vertex v_i, being v_i a candidate vertex to be part of the influence space of a segment s_j.

Belonging test.

As we can see in fig. 3.5, because of the nature of the δ function, some vertex that are not part of a particular segment are marked as candidates; and therefore a discrimination of the segments in a candidate vertex v_i is needed. We apply the following test to select the segment s_j in the segment list Ls for each vertex v_i:

- For each segment s_j in the segments list Ls, we compute the angle θ_{ij} between the weighted normal np_i (the mean value of the sum of the normal of the vertex and its $1 - connected$ neighbors) and the vector



(a) Vertices of the right hip segment. (b) Right hip influence space(red color).

Figure 3.5: *Candidate vertices of the right hip segment.*

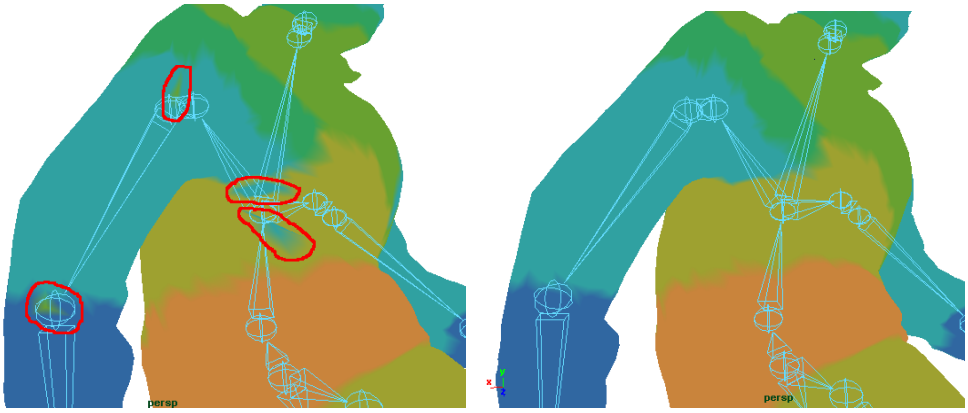
$s_j\vec{v}_i$ (the vector that had the shortest distance d_{ij} from a segment s_j to the vertex v_i)

- The segments with an angle $\theta_{ij} > 90$ between np_i and $s_j\vec{v}_i$ are discarded.
- End nodes cannot be discarded by the anterior rule.
- We assign the vertex v_i to the segment s_j with the lowest distance d_{ij}, measured from the segment to the vertex.

This simple set of rules allow us to apply our algorithm in meshes that are not in the ideal *T-Pose*, as can be seen in fig 3.7 it can be applied to rigged meshes with arbitrary poses.

Region merge.

As is shown in fig. 3.6(a), the vertex assignation with our *region-grow* algorithm and the *belonging test* can create "*false positives*" in meshes with arbitrary poses. This problem is caused due to the orientation of the pondered normal of some vertices with the vector $s_j\vec{v}_i$, and it depends basically of the face orientation in some vertices. We solve this problem using region-merging. Our region-merging method creates for each region s_j(corresponding to a segment), a list with subsets of vertices connected;



(a) False positives in a segmented mesh. (b) Segmented mesh after merge region.

Figure 3.6: *Comparative between the same mesh before and after apply the merge region procedure.*

we basically create subsets of vertices interconnected in a segment region. The largest subset in the list will be the definitive set for the computed segment region s_j, the remaining subsets will be merged each one with its largest neighbor region (the region that had the highest number of vertices connected with the analyzed subset).

The complexity of our segmentation method is $O(Sn^2)$, being n the number of vertices in a $3D$ mesh, and S the number of segments created from a skeleton, the O complexity analysis of our method is included in section 3.2.

Algorithm 3.2.4 Watershed based segmentation algorithm.

1: **function** $getInitialSeed()(inputMesh)$ Selects a vertex $iVtx$ from the input mesh, to be the initial seed for the region grow algorithm.
2: **end function**
3: **for** $i = 0 \rightarrow nSgt$ **do**
4: $\quad RegionGrow(iVtx)$
5: **end for**
6: **for** $i = 0 \rightarrow nVtx$ **do**
7: $\quad VertexBelongTest(Vtx_i)$
8: **end for**
9: $RegionMerge(ouputMesh)$

(a) T-Pose. (b) Pose A. (c) Pose. B

Figure 3.7: *Segmentation in meshes with different poses.*

Complexity analysis of the segmentation algorithm.

The analysis of the complexity O of the segmentation algorithm 3.2.4, based in our code implementation is:

$\sum_{i=1}^{S} a_{1i} n_v \cdot a_{2i} n_v +$	Region Grow.
$\sum_{j=1}^{n} (a_{3j} v_j s + a_{4j} v_j s) + n_v S + \sum_{k=1}^{m} (v_k a_{5k} n_v) + a_6 n_v S +$	Vertex belong test.
$n_v + \sum_{l=1}^{s} (n_v a_{7l} \cdot a_{8l} n_v + a_{9l} n_v)$	Region merge.

where $0 \leqq a_1 \ldots a_9 \leqq 1$ are constants, n_v is the total number of vertex in a $3D$ rigged mesh, and S is the total number of segments produced by the skeleton bounded to the mesh.
The simplification of this formula taking some of the constants $a1 \ldots a9$ as 1 is:

$n\ 2_v S +$	Region Grow.
$3 n_v S + m n_v^2 + a_6 n_v S +$	Vertex belong test.
$n_v + n_v^2 S + n_v S =$	Region merge.

$$2n_v^2 S + n_v^2 + S n_v (4 + a_6) + n_v =$$
$$2(S+1) n_v^2 + n_v (S(4 + a_6) + 1) =$$
$$n_v^2 S + n_v S (b + \tfrac{1}{s}) =$$
$$n_v^2 S$$

Therefore, the complexity of our segmentation algorithm is $O(Sn^2)$.

3.3 Results.

Algorithms 3.2.1 and 3.2.2 are similar and therefore the output will be similar. In polygonal meshes that are in **T pose** and all its vertices are convex the output will be practically the same (fig. 3.3). The differences rises when a polygon mesh is in an **arbitrary pose**. When a polygon mesh is in an arbitrary pose, the complexity of the problem is greater; therefore the need to know if the line connecting a vertex with a link or a joint is more transcendent to know which joint or link is closest to the vertex. Results in both algorithms depend on three factors: the correct position of the skeleton within the mesh, the pose and the convexity of the target mesh. The correct position of a skeleton means that all the links and joints of an animation skeleton must be within the mesh. If any joint or link is outside the mesh the results are not correct; and the vertices will be assigned to another joint or link (the assignation will be different depending on which kind of segmentation algorithm we are using) because of the input skeleton's position (fig. 3.8.). Although this particular situation is minimized with the post process in algorithm 3.2.2 there are some cases that cannot be solved if the input skeleton is not properly assigned. Algorithm 3.2.4 is the one that

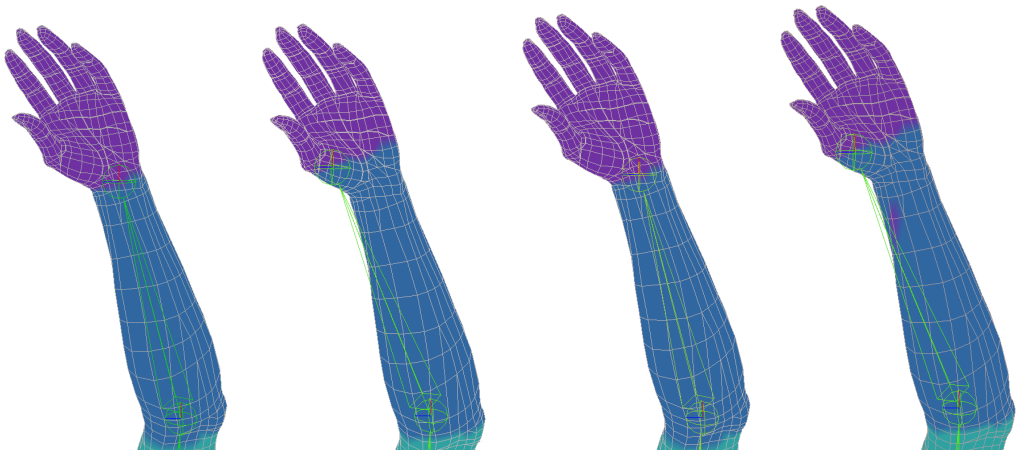
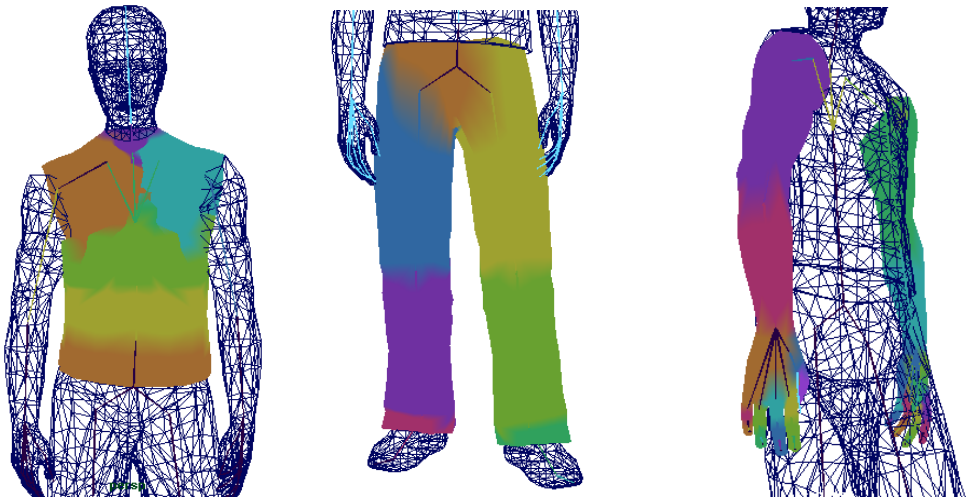


Figure 3.8: *Comparative of algorithms 3.2.1 and 3.2.2.* ***a*** *and* ***c****, joints correctly positioned inside the mesh;* ***b*** *and* ***d****, joints wrongly positioned(link outside).*

had better results in diverse situations, it can manage rigged-meshes with arbitrary poses, as well as rigged meshes in *T-Pose* or even meshes (with

its respective bound skeleton) that are not human-like as can be seen in fig. 3.11 ,where algorithm 3.2.4 is applied to $3D$ meshes that aren't human-like. algorithm 3.2.4 is also dependent of the skeleton position to have the best output, but their output quality is better than algorithm 3.2.2 because of its set of rules and the region merge post-process. The fact that it works solely with vertices of the rigged-mesh makes it a more flexible solution than algorithm 3.2.2, because it can manage meshes with any genus(not only closed meshes) and it can be easy adapted to rigged meshes with a single skeleton and multiple meshes(sub meshes, fig. 3.9). Their processing times are usually lower than algorithm 3.2.2 because it's based on the number of vertices instead of the number of voxels in the space; algorithm 3.2.2 can have low processing times if the space is voxelized at low resolution, but at expense of a lack of quality in the output segmentation.



(a) (b) (c)

Figure 3.9: *Segmentation applied over a multi-mesh character.*

Limitations.

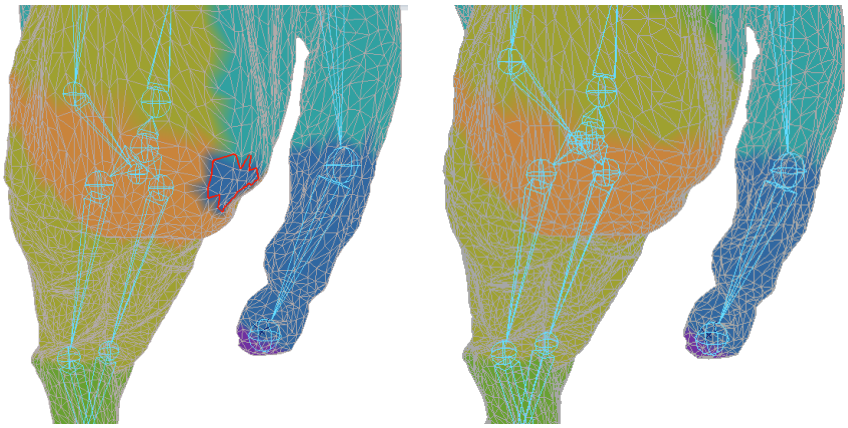
The algorithm 3.2.4 is the most flexible of the three algorithms, because it can manage an arbitrary pose. Algorithm 3.2.1 is the simpler one and is not dependent of the voxelization of the input mesh. Algorithm 3.2.2 without the post process part will give us poorer results in parts of the mesh that will be no problem in the algorithm 3.2.1 as can be seen in the little finger (fig.3.8 d) that is assigned to the link instead of the wrist joint (algorithm

3.2.2 assigns a vertex to the link of joint that had the shortest solid 3D line between each other). The errors in the assignation of vertices in algorithm 3.2.2 had a close relationship with the voxelization of the input mesh. The density or "resolution" of our voxelized mesh varies inversely in relation of the voxel size; therefore we can improve the results of the algorithm 3.2.2 by increasing the density of our voxelized space at exchange of computing time.

Density is also an influential factor at the time to convert the voxelized mesh to from a "*voxelized surface*" to a "*voxelized solid*". As can be seen in fig. 3.12, the step to transform a voxelized surface into a voxelized solid has had errors and the fingers of the left hand have been left as a hollow surface because of the voxelized space density. As a comparison, if we take the results over a mesh on *T pose* as the one in fig.3.3, and the output of the algorithm3.2.1 as the ideal case; the algorithm 3.2.2 without post process produces an error of 153 over 15576 (0.982 %) wrongly assigned vertices with a voxel size of 0.49 % of the total height of the mesh.
The post process in algorithm 3.2.2 is necessary because of the multiple errors generated with the voxelization process. If post process is introduced, the output in the case of fig.3.3 has an error number that goes from 153 to 43 wrongly assigned vertices (giving us a 0.27603% error over the ideal case). If we check fig. 3.4 *a* and *b*, we can verify that almost all the wrongly assigned vertices are in the eye balls, specifically in the right eye ball (the eye lashes that are a particular non convex area); with only one in the clavicle area giving us a more than acceptable result.

The results presented have been made with a *hLevel* parameter of one, the *hLevel* controls the deep level (up, and down) of search over the siblings of the vertex segment assigned, we suggest a value of *hLevel* between 1 or 2. Going deeper into the tree hierarchy can cause the opposite effect and assign a vertex to the nearest link; instead of making the assignation to the nearest link with a solid line between the vertex and the skeleton (an undesired case that happen in meshes with an arbitrary pose).

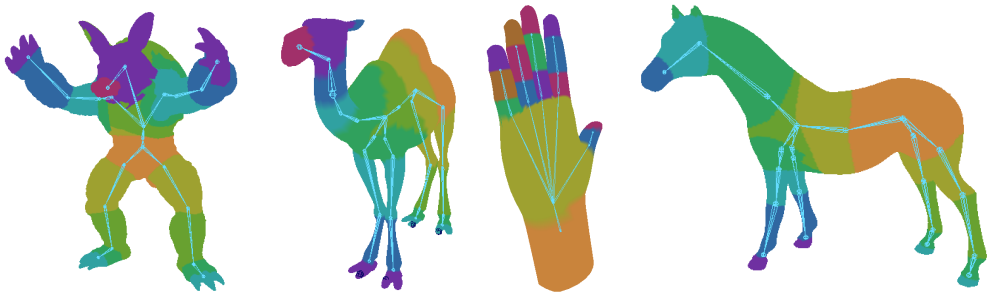
Algorithm 3.2.4 has some problems related to the post-process region-merge. In input meshes with low density of vertices (low resolution), a sub-region can be incorrectly assigned because of the low amount of data. The low resolution combined with the set of rules can produce isolated sub-regions of correctly assigned vertices that will be surrounded with another correctly assigned vertices. Therefore, when the region merge algorithm detects this sub-regions, they are deleted and assigned to the neighbor region that had more vertices in common. Since algorithm 3.2.4 uses Euclidean



(a) Segmentation algorithm. (b) Voxelized segmentation algorithm.

Figure 3.10: *Vertices wrongly assigned in segmentation algorithms 3.2.1 and 3.2.2. Red edges highlights the wrongly assigned areas.*

distances to measure the distance from a vertex to a segment, the position of the logic skeleton had a great influence over the output segmentation. In some cases it can produce false positives because the assignation depends on the shortest Euclidean distance, instead of having a skeleton mapping function over the mesh that could give us better results. Because of the

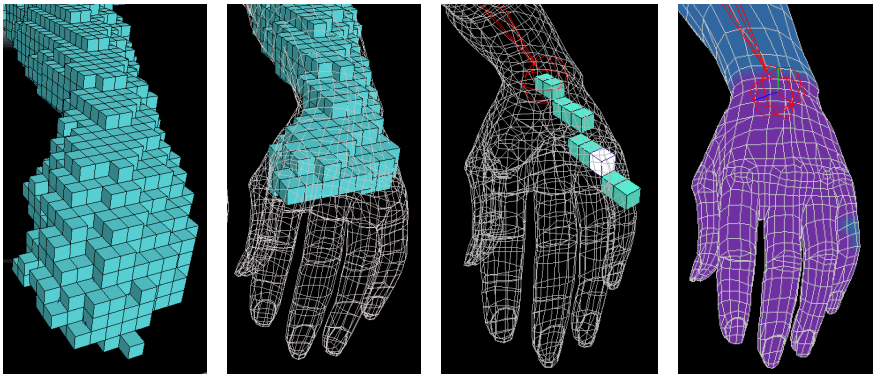


(a) (b) (c) (d)

Figure 3.11: *Watershed segmentation algorithm applied over meshes with different shapes and number of joints in their skeletons.*

limitations of algorithms 3.2.1 and 3.2.2; we suggest to apply algorithm 3.2.4 for an input rigged-mesh in any pose. Algorithm 3.2.2 can handle a mesh in an arbitrary position but is limited to closed meshes, depending on

the voxelization space density, and it is common that some vertices are not assigned properly. Although algorithm 3.2.2 had a post process to correct wrongly assigned vertices, some cases cannot be corrected and had to be properly assigned by hand. Algorithm 3.2.1 had good results with meses in *T-Pose*, but cannot manage arbitrary poses; it is also entirely geometric as algorithm 3.2.4 but their results are only comparable to algorithm 3.2.4 in that specific case.



(a) (b) (c) (d)

Figure 3.12: *Uncompleted filling in a hollow mesh* ***a*** *and* ***b****, generates a connection line with a null voxel.*

Chapter 4

Skinning.

Skinning is the process of changing the position of the vertex in a $3D$ object (usually a polygon mesh) in relation to the position of a **animation skeleton** (or logic skeleton). The skeleton scheme has the main advantage that it has no requirement specificity for each vertex on how to move, therefore the vertex movements are the result of an underlying skeleton movements. Only vertex associated to a specific link (or bone) will be affected by the bone's change of orientation or position (that is the reason why skinning is often related with skeleton-drive animation; although skinning methods based on cages does not depend directly on a skeleton). Skinning is used in fields related to computer animation: such as animation films, video games, and also in medical applications as a presentation tool. There is a great variety of methods to skinning a character (or $3D$ object); this methods can be classified in one of the next categories:

Linear Blend Skinning (LBS) or Skeleton Subspace Deformation (SSD).

In this deformation method, each vertex deformation (change of position) is the product of the sum of each joint in the skeleton multiplied by a weight. This technique is known in the literature by the name of *Skeleton Subspace Deformation* (**SSD**)[24], *Linear Blend Skinning* (**LBS**), *Enveloping* or *Vertex Blending*[11]. It has not been published properly, but it is widely used in video games and films since 1998 [23]. In **LBS** the vertices positions p_i are computed by a linear combination of rigid transformations (rotations and translations) of the influence joints j_k multiplied by a weight w_{ik} (where j is the vertex number and i is the skeleton's joint number). If a vertex has a weight $w_{j2} = 0$, this means that the joint 2 has no influence over the vertex j. It is desirable that the set of weights w_{ik} for each vertex is

normalized ($\sum_k^n w_{ik} = 1$); if it's not normalized, the possible rotations can produce serious artifacts related to magnifications of one of the joints j_k rigid transformations.

$$p'_i = \sum w_{ik} M_{\delta k} M_{Lk} p_i$$

A detailed discussion about the **LBS** will be elaborated in section 4.4. The main advantages of the **LBS** algorithm is its simple and efficient computation. This algorithm is used natively in professional animation software (such as Autodesk Maya, where it is called *smooth skinning*); but as it's known, it suffers from artifacts when some rotations are made by the influence joints, leading to the artifacts: "*collapsing elbow*" [24] and "*candy wrapper*" [11]. This well-known artifacts cannot be prevented by any means by a user painting the weights in the target mesh (usually the weight distribution is "painted" or assigned by an artist to achieve the desired effects. Automatic software fills the weighs in with an initial approximation). Since the artifacts are inherent to the **LBS** deformation scheme, an important number of works have been published to solve this problem; this methods can be categorized by the way they compute the influence weights for the *LBS* algorithm:

- **Example based:** The number of works developed within this category make it a populated one; in this category the most relevant works are: ([24],[46],[42],[14],[28],[27],[29]). In all the cited methods, the main idea is to compute the weights of a $3D$ mesh using a set of examples (a set of $3D$ meshes in different poses). New poses (poses that are not in the examples) will be the result of the interpolation scheme to compute the vertices position on the target mesh. The approach used in **PSD**, is to solve the skinning problem as an interpolation one [24]; they take the **LBS** main equation and substituted it by a radial basis equation system, which is similar in essence to **LBS** because it's also a linear combination; but **PSD** is a linear combination of nonlinear functions of the vertex's distance (a Gaussian radial basis function).
  The method known as **MWE** [46] is an extension of the **LBS** algorithm; in this method instead of having one weight per rigid transformation in the skinning's main equation, a weight is assigned to each element (rows and columns) of the rigid transformation matrix (the matrix obtained by the product of the $M_{\delta k} M_{Lk}$ matrices). The main advantage of having a weight for each element of the transformation matrix is that the influence of the weights is not limited only to the rigid transformation as a single operation; but each part of the rigid transformation can be assigned with a different influence weight to

achieve the desired deformation. The weights values are computed by a minimum square lineal system using a set of input poses as deformation examples. A similar approach to **MWE** is [28], where a set of input meshes and skeletons in different poses are called examples. The examples can or cannot represent a realistic pose; the objective of this examples is being a tool to define the degrees of freedom of each influence joint, and for every vertex in the target mesh to find their influence joints. This is done by solving a bilinear minimum square system to find the desired parameters. This method is an extension of **LBS** with the particularity that new joints are added to the original skeleton. This new joints are created to allow new deformations to the target mesh; the new joints are added automatically to the target skeleton, but users are allowed to modify the final skeleton. A more recent approach is the work described in [14]. The main idea of this work is to use proxy bones to compute the weights of the target mesh. The proxy bones are a division of the $3D$ mesh that are created through a set of meshes; this meshes are the target model in different poses. The division of the target mesh is made by grouping a set of triangles that has the same rotation sequence within a time line from a starting position. Once the target mesh has been divided in influence areas, influence areas are assigned for each vertex. Influence areas had the same function of the skeleton's links (no skeleton is defined in this method; and the influence areas are used to create an underlying skeleton). The joint's weights are computed by using two methods of minimum squares; for poses defined within the example meshes **TSVD** is used; otherwise **NNSL** is the chosen method.

- **Function based:** This methods had two modalities:

  1. Based on compute automatically the weights of the **LBS**.
  2. Based on replacing partially or completely the blending method (substituting the rigid transformation matrix with a different rigid transformation tool), or adding a correcting method for the deformation achieved with the **LBS**.

  In the first category we found [49]; its main approach uses a non-linear model to compute the weights of the **LBS**. The computation of the weights is calculated using a polynomial function that is based on a quantity called *influence ratio* r. The ratio r is computed using the angles between the skeleton's joints in their initial pose and a vertex point p; r needs the arbitrary constant α to be defined by the user.

Then r is the argument of a *weight distribution function*(the polynomial $W(r) = \sum_{i=0}^{d} a_i r^i$) that is used to compute the value of a weight w_{ji} of the vertex p_j in the joint j_i. In [3], a function based on heat diffusion ($-\Delta w^i + H w^i = H p^i$) is used to compute the weights of the **LBS** algorithm; where Δ is the Laplacian of a discrete surface, p^i is a vector where the element $p_j^j = 1$(rigid skinning) if the nearest bone to vertex j is i and $p_j^j = 0$ otherwise. H is a diagonal matrix which will have in H_{jj} the closest weight contribution to the vertex j.
One of the latest methods to compute weights automatically is [13], where the weights are computed by using a Laplacian energy function subject to an upper and lower bound constraint minimizer. Solved through a *Finite Element Method (FEM)* where the weights w_j are the elements to be minimized and computed following a set of desirable properties; this work is relevant in particular due to its general approach to the problem. The weights are values that depend of the elements called handlers; the handlers can be the elements of a cage (in $2D$ or $3D$) or the joints in a skeleton. Therefore it can be applied to images or skinning in $3D$ cage-based or skeleton-driven animation. The same authors later proposed a method in [12] where new weights are appended to the existing ones adding weights generated with isotropic B-spilne functions in weight space centered at some seed locations. The seeds are spread using a discrete multi-dimensional version of optimized farthest points, improving the deformation behavior of the models where the method is applied; specially the ones with small number of original weights.

In our second category we find methods like [23], where we find a change in the interpolation method; from **LBS** to *spherical blend skinning* (**SBS**). **SBS** uses quaternions to blend the final position of a deformed vertex; all the matrix in the **LBS** are changed to its equivalent in quaternions. For each vertex influenced by the set of joints $j_1, ..., j_n$ a rotation center r_c is computed; the rotation matrix defined as Q is used in quaternion linear interpolation (**QLERP**) between multiple quaternions (which interpolate the rotation part). The used formula is: $v' = Q(v - r_c) + \sum_{i=1}^{n} w_i C j_i r_c$; where the matrix C_j is the Matrix used in the **LBS** algorithm. Inspecting the formula used in **SBS** is easy to see why **SBS** is slower than **LBS**; basically It's the addition of two position vectors: the vector obtained with the **LBS** method applied to r_c(computed with a based least squares optimization method *SVD*), plus the position vector from the **QLERP**

applied to the vertex in a local reference frame defined by r_c.
A more sophisticated approach is introduced by the author of [23] in [21], where dual quaternions are used to solve in an efficient way the well-known problems of **LBS**. Dual quaternions are quaternions whose elements are dual numbers ($\hat{q} = \hat{w}+i\hat{x}+j\hat{y}+k\hat{z}$). Dual numbers are similar to complex numbers ($\hat{a} = a_0+\varepsilon a_\varepsilon$) with the property $\varepsilon^2 = 0$ and a complex numbers analogous conjugation. Dual quaternions had a defined set of rules and algebra (that is similar to conventional quaternions). The advantage of using dual quaternions as interpolation tool is the possibility of perform interpolations in a simplified, elegant and efficient way (38% slower than **LBS**). The interpolation methods in dual quaternions are based on their quaternions counterparts, without the problem of solving a different rotation center for each vertex (dual quaternions can also represent $3D$ translations that are interpreted geometrically as screw motion). Because of the equivalence of operations between quaternions and dual quaternions, a version of **QLERP** for dual quaternions is made (called Dual quaternion linear blending **DLB**) $DBL(w; \hat{q}_1, ..., \hat{q}_n) = \frac{w_1\hat{q}_1+...+w_n\hat{q}_n}{\|w_1\hat{q}_1+...+w_n\hat{q}_n\|}$. One problem with **DLB** is that its interpolation is not linear; and could, in the most extreme cases, produce a difference of 8.15 degrees and a 15 % of the translation with the linear interpolation of dual quaternions (**ScLERP**). If a more exact approximation is needed, the authors have proposed an algorithm that can be controlled by a precision parameter(in animation such precision is not needed) called Dual quaternion iterative blending **DIB**. Interpolation with **SBS** and **DQ** eliminates the candy wrapper artifact and produces better results than the **LBS** algorithm. In [50], the candy wrapper artifact is corrected by computing an additional weight $\delta\alpha_i$ per vertex; based on the angles in animation at binding time. $\delta\alpha_i$ is used to compute a rotation matrix (restricted to the X canonical axis in the paper) that is multiplied prior to $M_{\delta k}$ in the **LBS** equation; also an operation over the vertices to compensate the collapsing joint is introduced. This operation consists in choosing a collapsing joint; then the vertices affected by this particular joint will be re-computed by a stretch operation that is basically a *vector length compensation* from the chosen joint to the affected vertex. This is done by taking the length of the deformed vertex as the base; then an extra length is added based on the distances and weights from the influence joints to the deformed vertex. One of the methods that adds a post processing to the **LBS** is [39] which adds a volume correction stage after the skinning deformation of a target mesh. The volume correction is treated as a minimization

problem of a correction vector u that is computed using Lagrange multipliers. The volume change of a mesh after its deformation is defined as $\Delta V = V(p) - V(\bar{p})$ where p is the surface of the deformed model and $\bar{p}$ is the original one. The Lagrange multiplier used to solve this problem is: $\Lambda(u, \lambda) = \|u\|^2 + \lambda(V(p+u) - V(\bar{p}))$, satisfying $\nabla\Lambda(u, \lambda) = 0$. To achieve a uniform correction by the vector u the process is divided in 3 steps (one step per coordinate axis). Where the correction is made by a percentage of $\mu_n; n = 0, 1, 2$ per axis. Along with the set of weights μ, a set of weights γ are used (the γ set of weights are applied directly to the u vector along with the weights μ at the moment of solve the Lagrangian) to achieve a localized deformation effect applied to each local frame joints. A similar method can be found in [44], which is also a post process to correct the deformed volume obtained with **LBS**. To compute the original mesh volume, it uses signed tetrahedron volume created between each of the input mesh triangles and the origin. The change of volume is computed by: $vol(P' + \lambda \circ V = vol(P)$ where P are the mesh vertices; V is a displacement vector field and λ is a scale factor applied to V. The last equation can be written as an implicit third grade equation where the independent variable will be λ. The displacement vector field will be computed (a blend of the vector computed between a vertex and the joints) based on a δ function. By solving λ's value, the correction of volume is performed over the deformed mesh. This method uses a set of new weights S to control the correction in a localized level. S is a set of weights multiplied to each of the vectors in V (a weight s_i is defined for a vertex p_i) to achieve custom effects like muscle bulging. The set S can be painted by an artist, or can be computed automatically. This method can only deal with a deformation at time because the correction performed is global; therefore, the correction must be applied for each joint one after the other in a hierarchical order to preserve the volume locally. The two previous methods share a feature that can be considered as a major drawback: they cannot correct the candy wrapper artifact when the twist angle is of exactly 180° and all their multipliers; this is due to both methods were designed as a correction stage, but when the twist angle value is 180°, the trajectories of the points cannot be corrected because some vertices will intersect in one point position inherent to its linear behavior. Two of the latest methods are [20] and [22]. In [20], blending bones are used to approximate non-linear skinning with a set of weights computed specifically to work with the extra blending bones. The blending bones approximate the deformed output of a non-linear

deformer using a precomputed deformer output. The precompute is calculated as a minimization error optimization method, that takes the desired deformer as an input in some key frames, and computes the weights of the blend bones to minimize the error of the output vertex's position.

In [22], a different approach is taken by combining two deformers. For twist rotations they use a quaternion based deformation solver, and linear blend deformation is used for any other kind of deformation. Each deformer had its own set of weights; the weights are computed and optimized using examples for some representative poses using bi-harmonic weights as base.

For an easier understanding and comparison of some of the main skinning methods, we have created the comparative table 4.1.

Features	[49]	[3]	[13]	[21]	[23]	[50]	[44]	[39]	[20]	[22]
Non-linear Polynomial weight func.	√	√	√	×	×	×	×	×	×	×
Heat-diffusion weight function.	×	√	×	×	×	×	×	×	×	×
Rigid skinning pre-proc.	×	√	×	×	×	×	×	×	×	×
Quaternion based.	×	×	×	√	√	×	×	×	√	×
LBS Matrix modification.	×	×	×	×	×	√	×	×	×	×
Post-proc. Volume correction.	×	×	×	×	×	×	√	×	×	√
Non-lin. aprox. using addnl. bones.	×	×	×	×	×	×	×	√	×	×
Two deformers combination.	×	×	×	×	×	×	×	×	√	×

Table 4.1: *Weight distribution and Skinning methods features comparison. Methods: RSDSD [49],Pinocchio[3], BBW[13],DQS/DIB [21],SBS [23],Strech-it [50],VPMS [44],EVPSSC [39], ALNS [20], EIDCA [22]*

Cage Based Skinning.

Cage based skinning is a general technique were the objects are deformed by embedding them into an object that has a simpler structure. The control volume is usually defined using a structure with control points (the cage). The spatial deformations are achieved through the manipulation of control points because the vertices of the object to deform are mapped to the control object vertices.
An influential work is: [18] called Free Form Deformation (**FFD**). It was originally presented by Parry and Sederberg in 1986 as a general technique where objects are deformed by warping a volume space where the objects are embedded. **FFD** share some similarities with **LBS**; both use weights to control the number of influence that had each of the control vertices in **FFD** (the influence joints in **LBS** will be their equivalent) over the target's mesh

vertices. **FFD** uses a "cage" to control the deformation that is basically a low resolution version of the target mesh. The deformation algorithm has 3 phases. First: the cage is sculpted and bound to the target mesh. Second: the vertices of the cage are registered (an influence weight is computed) based on the Euclidean distance from the cage vertices to the target mesh vertices that can be modified manually by the user. Third: the deformation phase, which is computed with the equation: $P_{def} = \sum_{k=1}^{n} u_k^P P_k^{def}$. Where P_{def} is the deformed vertex, u_k^P is the weight for a specific control element k and $P_k^{def} = P_k^R Q_k^D$ and Q_k^D is the transformation matrix corresponding to the local coordinated system of the control element k of the cage; P_k^R is the representation of the vertex P in the local coordinated system of the control element k on the original cage R. The parameters such as u_k^P can be edited to obtain better results and avoid artifacts; the technique was implemented in *Maya 2.0*.

In [16] *mean value coordinates* (**MVC**) are applied to skinning using a cage; this new interpolant was proposed by Floater [9] to generate smooth coordinates for star-shaped polygons (later it was demonstrated that **MVC** generates smooth coordinates for any simple polygon). The coordinates are defined as the set of weight functions: $w_j = \frac{\tan[\frac{\alpha_{j-1}}{2}]+\tan[\frac{\alpha_j}{2}]}{\|p_j - v\|}$ For a polygon with vertex p_j, a point inside this polygon can be expressed as the linear combination of this weights. The $3D$ generalization consist in projecting each triangle of a triangular mesh over a unit sphere; a set of weights w_i per triangle are computed creating a system that depends on the normalized set of weights of all the triangles in the target mesh.

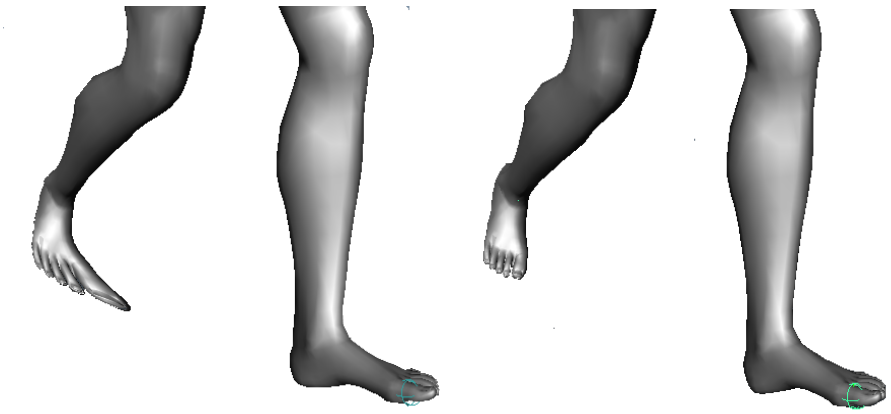
To apply **MVC** in skinning, the target mesh weights are precomputed before any deformation; then taken as a constant. The deformation is made by computing any change in the position of the cage vertices as a linear combination. The main problems in **MVC** are: negative weights and absence of locality. Negative weights can produce artifacts; when a vertex of the cage is deformed, a close vertex of the target model is affected due to its negative value. The absence of locality is present in this artifact because a vertex of the cage had influence over a vertex of the target mesh that it supposed not to have. This particular problem has been solved in [25]; where all the weights values of the affine combination are positive. This is achieved by projecting a sphere into the cage instead of projectting the cage into the sphere. Computing the weights of **PMVC** is more time consuming than **MVC**, and to accelerate this process **GPU** computing is used to compute the cage weights. The result is a better framework to cage deformations without the artifacts of the original **MVC**.

An application of the **PMVC** cage deformation is found in [17]; the cage

based deformation is taken as a base and adapts it to skeleton driven skinning. The basic idea is to apply a variation of **LBS** skinning to the cage in what they call *templates*. The templates are specific behaviors to deform the cage based on the part where the cage vertices had influence. The cage vertices that are over a skeleton joints are deformed by the *joint template*; the *joint template* deforms a vertex in a spherical way. The vertices that are between two joints are deformed by the *bone template*; *bone template* vertices are deformed in a cylindrical way with a skew parameter. Therefore the cage is skinned, the skeleton operates over the cage, and the cage operates over the target mesh vertices using **PMVC** deformations. To obtain custom deformation behavior; an additional pose needs to be defined. A rigid transformation matrix is computed (and later multiplied to the template matrix) to adjust the deformation of a cage to the desired one. An automatic cage fitting is used to obtain the first approximation of an arbitrary model cage; then a user manually modifies the model fitted cage to obtain the desired deformation behavior. *Harmonic Coordinates* (**HC** [7]) Harmonic coordinates are a cage based deformation method, based on Laplace equation $\nabla^2 h_i(p) = 0, p \in Int(C)$; where $h_i(p)$ is a weight function defined in a close cage C with vertices C_i; with $\sum_i h_i(p) = 1$ for all $p \in C$. To solve the weight values, a hierarchical finite difference solver is used by applying a grid division of the space to allocate the cage; then the cells of the grid are tagged by their neighborhood and intersection with the cage. Then the weight values are computed to the vertices of the cage and a Laplacian smoothing is performed by iteratively computeing the weight value of the interior cells (points of the target mesh); until the average change of a cell drops below a specific threshold. An elegant and sophisticated cage based skinning deformation are the *Green Coordinates* (**GC**). Based on the green theorem where a green function is a fundamental function of the Laplace equation (therefore is a harmonic function); the **GC** are defined as $\eta \mapsto F(\eta; P') = \sum_{i \in I_{\mathbb{V}}} \phi_i(\eta) v'_i + \sum_{j \in I_{\mathbb{T}}} \psi_j(\eta) s_j n(t'_j)$. Where P' is the deformed cage, η is an interior point (a point of the target mesh), v'_i and t'_j are the vertices and faces of P'. One main feature in **GC** is shape preservation, achieved by combining the vertices and the normals of the cage faces; where the exact relation is coded in the coordinate functions ϕ_i, ψ_j. The scalars s_j are used to control the *stretch* that the faces t_j undergo as the cage is deformed. Another feature about **GC** is that it allows the use of partial cages. Partial cages are cages that only had effect (or deform) the vertices of the target mesh that are inside the partial cage, leaving the exterior vertices with a smooth transition of the deformation of the partial cage. As **PMVC** and **HC**, **GC** are computed once and then stored to compute the deformation at interactive levels.

4.1 Segmentation and its application to skinning.

One of the main applications of the segmentation algorithms described in section 3 is to generate weights automatically for the LBS skinning algorithm. The weights for the LBS algorithm can be generated by different approaches such as ones mentioned in section 4. We use our segmentation algorithm as a base to compute the weights values. The main advantage of generating weights based on the segmentation of an input mesh is that we have identified the main influence joint for each vertex in a mesh, even if that mesh is in an arbitrary pose. Having assigned the vertex to a specific joint(the base of a segment) allows us to generate weights in a hierarchical way, involving only the joints that were directly related to the main joint; instead of distributing the weights with a geometric method (the common approach), where the joint's influence is calculated using their distance to a vertex. As an example, we can see that *Autodesk Maya's* automatic weight algorithm generates artifacts because, apparently, the weights are calculated using the measured euclidean distance from a sphere centered in a joint to a vertex within a specific radio,



(a) Autodesk Maya automatic weight. (b) Segmentation based automatic weight.

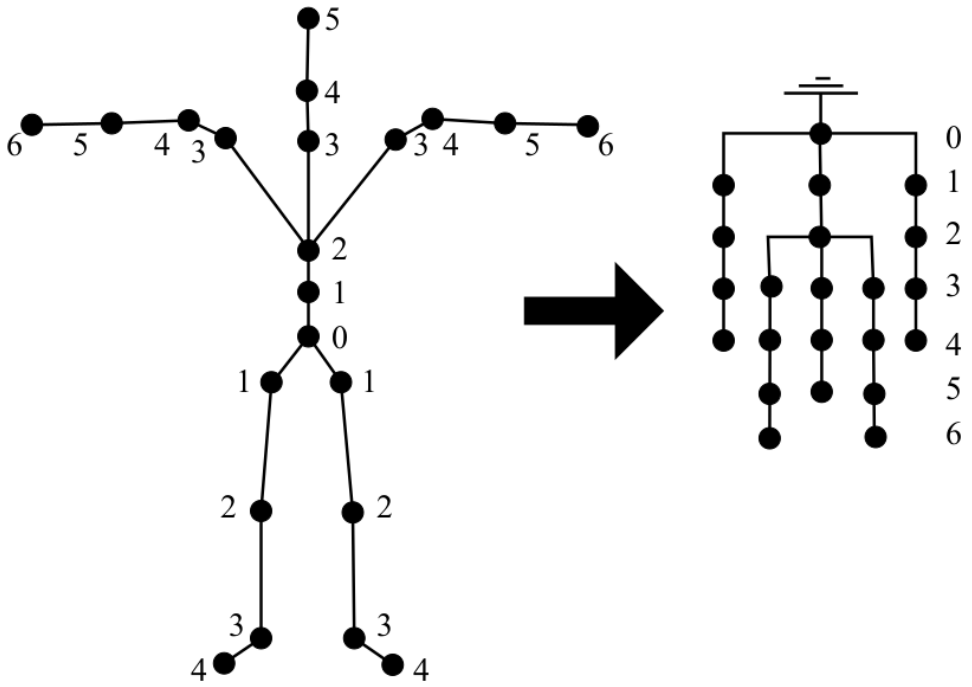
Figure 4.1: *Artifacts generated in an animation frame of a mesh due to a improper weight assignation by the Autodesk Maya automatic weight assignation algorithm.*

creating an artifact that is the direct result of an improper weight assignment caused by the proximity of joints (an example can be seen in the

artifact generated on feet of the model in figure 4.1).

4.2 Segmentation based weight assign algorithm.

Our weight assigning algorithm is based on the segmentation algorithms described in chapter 3. For each vertex v_i we store in a data structure the main influence joint j_k, then using a metric distance (it can be euclidean, geodesic, or something else), the weight for the main joint an its "siblings" are computed based on a chosen distribution function $w_{i_k} = F(d_{i_k})$. The term siblings is used as a reference to the child nodes for the parent node on the hierarchy tree within a logic skeleton, the equivalence between a logic skelton and its hierarchy tree is showed in figure 4.2. As an example: if the



(a)

Figure 4.2: *Equivalence between a logic skeleton and a n-ary hierarchy tree.*

hierarchy h_{v_i} of the skinning weight assigning algorithm is 0; the algorithm will be applied only on the joint that is assigned as the main influence joint for each vertex v_i. If $h_{v_i} = 1$, the weight assigning algorithm will

be applied, if possible, over the direct child and parent nodes of the main influence joint. In the same fashion, if $h_{v_i} = n$, the weight algorithm will be applied iteratively over the sibling nodes of the main influence node.

Algorithm 4.2.1 Automatic weight assigning algorithm.

```
1: for x = 0 → nVtx do        ▷ where nVtx is the total number of vertices in a mesh.
2:     cVrt ← Vert(x)                                  ▷ current vertex in the loop.
3:     pnt ← GetPos(cVert)                  ▷ current vertex's position in the space.
4:     mNod ← GetMnJnt(cVert)
5:     cmpStWght(dstbFn, dstFn, mNod, pnt, cVrt)
6:     for y = 0 → nhrchy do ▷ where nhrchy is the number of levels up and down in
   hierarchy.
7:         pNod ← GetPrntNod(mNod, y)      ▷ gets the parent node of mNod in the
   upper level y.
8:         ndLst ← GetChlds(mNod, y)       ▷ gets the children nodes of mNod in the
   lower level y.
9:         pushLst(ndLst, pNod)
10:        szL ← SizeLst(ndLst)
11:        for z = 0 → szL do
12:            cNod ← getElmLst(ndLst, z)
13:            cmpStWght(dstbFn, dstFn, cNod, pnt, cVrt)
14:        end for
15:    end for
16: end for

function cmpStWght(dstbFn, dstFn, nod, pnt, vrt)
    dElm = dstFn(pnt, nod)
    wght = dstbFn(dElm)
    setWeight(wght, vrt, nod)
end function
```

The function *cmpStWght* computes and stores the weight value for a specific vertex; using *dstFn* as the function that calculates the distance from the vertex *vert* to a joint *nod*. Finally the weight is calculated by the distribution function *dstbFn* and it is stored into a data structure.

4.3 Distribution Function and Distance function.

In general, the algorithms that calculates automatically weights for the LBS had two components:

1. *Distance function* or $DstF$. A function that calculates a value. This value can have a direct or indirect relation with a kind of distance

from a vertex to the main influence joint (and consequently the main influence link).

2. *Distribution Function* or $DtbF$. This function takes a set of values, and maps it to a range between 0 and 1. Its output will be the weights assigned to a vertex for each joint of the character. If the sum of the weight values is either more or less than 1 the produced deformation will have artifacts depending on the influence of each joint.

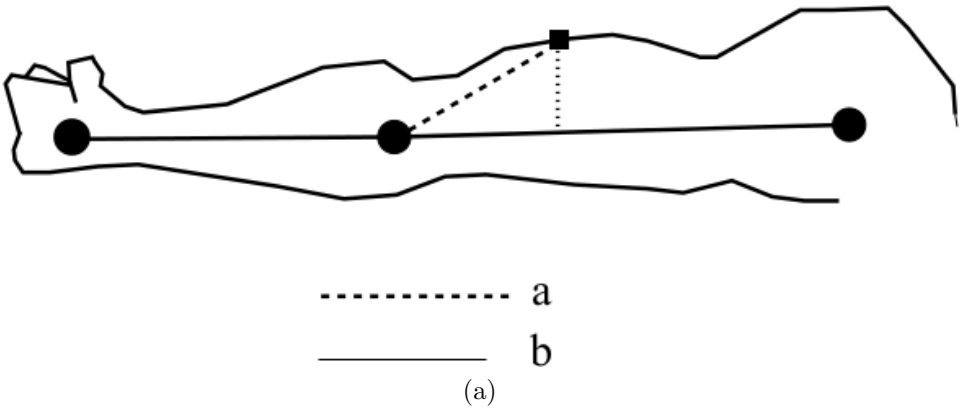
The $DstF$ and $DtbF$ generates weights that will produce rotations with general behavior, because are applied to all the parts of the body indistinctly. The works made by [39] show that effects can be added by computing the weights with a distinct distribution function than other parts of the mesh, using a method (like assigning manually the distribution function to a set of vertices or using precomputed templates) that will modify the behavior of the vertices where the distribution function was applied.

Distance function.

The distance function ($DstF$) is important because its output will be used directly by the distribution function. Therefore, a function that calculates the Euclidean distance will give us a different output than other one that uses a geodesic distance. We can find an example of an indirect $DstF$in [49] instead of using some kind of measure they compute r, which is an influence ratio that involves the angles between the main influence links of a vertex and the vector formed by the vertex and the main influence joint. Therefore r has an inverse proportional relation with the distance from the influence joint to the vertex. In our particular case, we use the δ function described in 3.1 as $DstF$ because its output is a normalized measurement based on distance of the projection of the vertex over the link; instead of the euclidean distance from the vertex to a joint, that depends on the shape of the input mesh (fig. 4.3).

Distribution Function.

The distribution function ($DtbF$) is the most important part in the weight calculation for the LBS algorithm; the deformation behavior of a mesh depends entirely on the chosen distribution function. Any function can be used as distribution function, but the quality of the output deformation will depend on this function. In [49] their $DtbF$ is: $W(r)$, where r is the



(a)

Figure 4.3: *Comparative between distances. a) euclidean distance. b) projection over link distance.*

product of an indirect $DstF$. The distribution function $W(r) = -6.4r^5 + 16r^4 - 14.8r^3 + 6.2r^2$ is a polynomial function of fifth degree to ensure smoothness, continuity and symmetry. This equation was calculated by a set of linear equations that satisfy a set of constraints. In [3], a heat equilibrium equation is used as $DtbF$; $\frac{\partial w^i}{\partial t} = \Delta w^i + H(p^i - w^i) = 0$ where Δ is the discrete surface Laplacian, p^i is a segmentation vector and H is a diagonal matrix of $n \times n$ degree(being n the number of vertexes in the mesh), where H_{jj} is the heat contribution from the nearest bone to a specific vertex. H_{jj} uses the Euclidean distance as $DstF$ combined with a constant value $H_{jj} = \frac{c}{d(j)^2}$ if the shortest line segment from the vertex to the bone is inside of the volume, $H_{jj} = 0$ If not, in [39] uses volumetric geodesic distances as an initial value, like $DstF$ and $DtbF$; but within its method of volume correction they use Gaussian and sinusoidal function as $DtbF$ to achieve effects on specific parts of a character's body. We currently use the most widely $DtbF$ used; which is a Gaussian function defined as:

$$f(x) = ae^{-\frac{(x-b)^2}{2c^2}} \tag{4.1}$$

where:

- The constant a is usually one.
- The variable x is the output value from the distance function.
- Constant b will be the place where the value will have its greatest value. Usually is in the center of the link where the vertex is assigned.

- The inflexion point is controlled by the value of the constant c and it is important because if we put a very low or a very high value, we might have rotational continuity artifacts.

After assigning the weight value to the joints that had influence over a vertex, we normalize the values. If the weight values are not normalized, artifacts are produced in a vertex as result of the sum greater than one in the weight values of the influence joints.

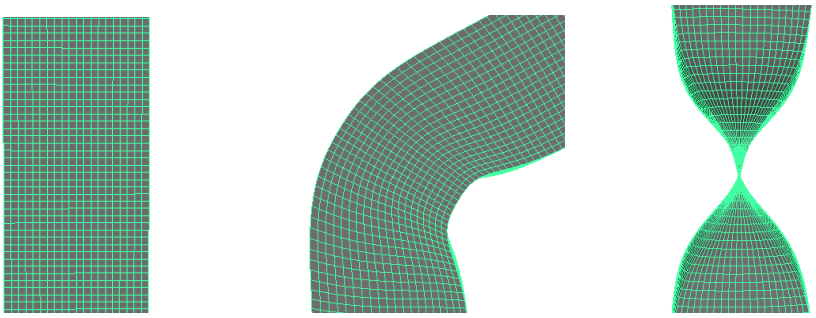
$$wT_i = \sum_{j=0}^{n} w_{ij}$$

$$wN_{ij} = \frac{w_{ij}}{wT_i} \tag{4.2}$$

In our test over multiple meshes we had used a hierarchy number of one. For most of the vertices this produce a three influence joints for each vertex; which is the number of influence joints that is regularly used for the vertices of a rigged character. The values used for the Gaussian function are $a = 1.3$,$b = 0.5$ and $c = 0.25$, this parameter values have produced the best results in our tests.

4.4 Segmentation based LBS Skinning.

Linear Blending Skinning is the most common skinning algorithm used in industry nowadays. The technique was unpublished as it's stated in [24], but is commonly used as a base of more general or complex deformation schemes. One of the main problems of this widely used deformation scheme is the "candy wrapper" artifact. The candy wrapper artifact is generated when a link rotates more than 60°, and is at its max when the rotation reaches over 180° in the axis that is aligned with the link direction (a twist rotation). The candy wrapper, in plain words, is a loss of volume over the mesh caused mainly because of an abrupt change of position between two sets of vertices when this two sets are part of a neighborhood in a surface (fig. 4.4 (c)). Specifically this artifact is generated when a joint rotates over the axis that is closest to the direction of the link created by the joint that rotates and the previous joint in the hierarchy. To fully understand why the candy wrapper artifact is created, we will give a short explanation of how the LBS scheme works.



(a) Original mesh. (b) Rotation of 60° in the x axis. (c) Rotation of 180° in the y axis.

Figure 4.4: *Linear Blending Skinning deformation method.*

LBS Skinning deformation scheme.

The LBS Skinning is basically the sum of a set of vectors for each vertex of a polygonal mesh defined by the formula:

$$p'_i = \sum w_{ik} M_{\delta k} M_{Lk} p_i \tag{4.3}$$

where w_{ik} is the weight defined per each joint k in the skeleton, and only the joints that had influence over a vertex can be greater than zero. $M_{\delta k}$ is the rigid transformation matrix in world coordinates for a joint k in the current position of the skeleton, and M_{Lk} is the rigid transformation matrix used to transform from world coordinates to local coordinates for a joint k. In detail, for each joint j_k in a vertex p_i, the operation is: $M_{Lk}p_i$ transforms, from world coordinates, the point p to the local frame defined by the joint j_k, resulting in the position of p_i in the local coordinates of j_k. Then this local position is transformed from the current frame position of the skeleton by the multiplication of $M_{\delta k}$ rigid transformation matrix, a common example of a rigid transformation matrix will be $M_{\delta k} = Mr_k * Mt_k$. In this particular case, the rigid transformation is a rotation followed by a transformation. The operation $M_{\delta k}M_{Lk}p_i$ will give us as result a vector that is finally scaled by the multiplication of the weight factor w_{ik} as can be seen in figure 4.5.

The LBS Skinning is a simple and low cost scheme of deformation in floating point operations. It can produce good results for deformations in the limbs of a character in skeleton driven animation, depending on the weights assigned for each vertex. When a twist rotation is made, the mesh tends to lose volume due to the addition of vectors with opposite direction

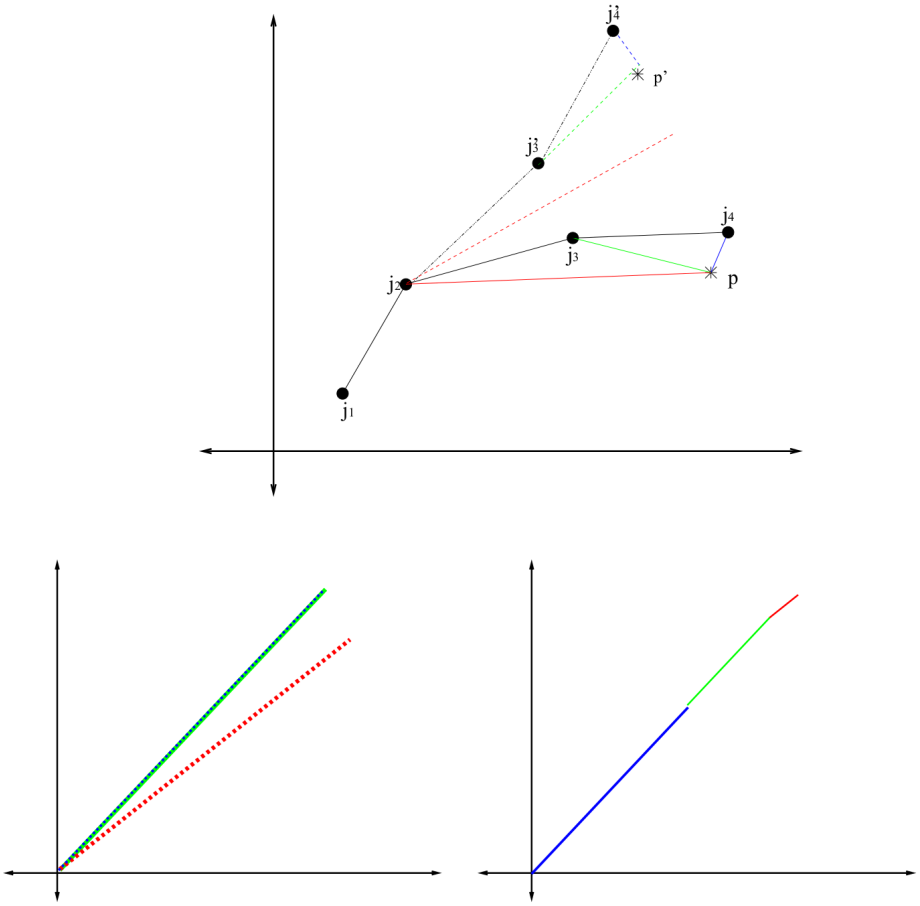


Figure 4.5: *Skinning process for a vertex p_i. Up: Operations in local frames of joints j_k.Left: Unscaled position for each joint j_k. Right: Result of the sum of the scaled points for the influence joints j_k.*

in their j_k local frame as is showed in fig. 4.6 (a). Because of the vertices position, the reduction of the volume forms concentric squares that reduce its area until it becomes zero when the weight w_k from the joint that was rotated and its predecessor w_{ik-1} had approximately the same orientation.

The candy wrapper artifact can be compared with the action of having two frames of rigid material with a set of threads attached to each one. Therefore when one of the frames is rotated 180° in the axis that had the same direction of the threads that joints the frames(linear rotation trajectories); they intercept each other in the center fig. 4.6 *b*. This behavior is created by the difference between the vectors added by the linear blend in equation 4.3. The vectors with approximately the same weight value in

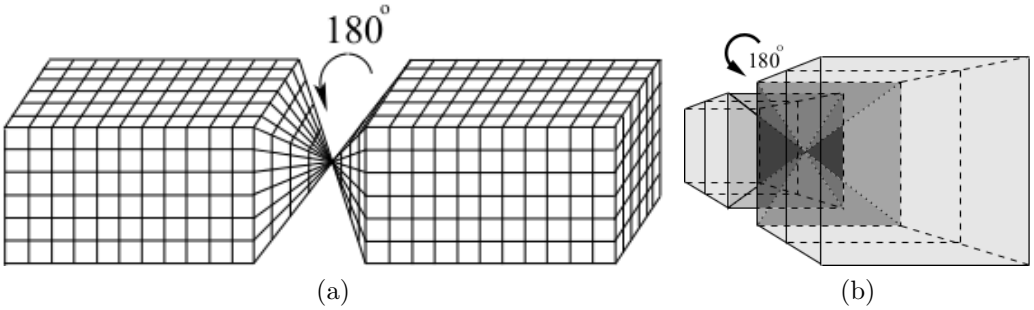


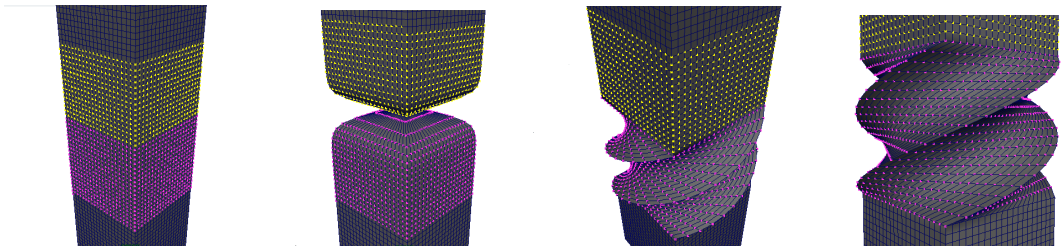
Figure 4.6: *Candy wrapper artifact.*

the influence link (the two most influential joints for the vertex v_i) will be the vertices that had the most volume lost. Because of this behavior, there is no set of weights that can achieve a volumeless deformation in LBS; the angles that are multiples of 180 will produce the artifact due to the indirect subtraction of vectors because of the rotation around an axis.

Our approach.

To eliminate the loss of volume in a twist rotation; we have identified that an abrupt change of rotation angle in the joints which are part of the axis in the closest link to a vertex is the main reason of the loss of volume in LBS. Our approach is based on keeping the same twist rotation angle over the link of all the influence joints in a vertex when a rotation is applied. Although the angle will be the same for all the joints in a vertex; this will progressively change depending on the position of the computed vertex and its projection on the link segment (the closest link). This value will be computed using the δ function.
In our modification over the LBS deformation scheme, we use the segmentation algorithm; therefore, two consecutive joints make a segment(a link in skeleton drive animation). A link had two joints: joint a is the vertex with lowest hierarchy and b will be its "child" joint. In our rotation scheme we apply the rotation over the link axis only to the vertices of that specific segment; rotations over the joint a will be applied as normally in the LBS algorithm. When a rotation is made over the joint b; we compute the rotation angle progressively for each vertex on that specific segment. As can be seen in fig. 4.7 when we make a 180° rotation over its link axis; the LBS applies the deformation in two segments (fig. 4.7 a); but our deformation scheme is applied only in one segment (fig. 4.7 b). We believe that this is a more natural behavior if we take the way of how a human limb deforms

itself.



(a) (b) (c) (d)

Figure 4.7: *Deformation of a bar with the LBS (b), and our approach (c),(d).*

Simplest case.

To explain our modification to the LBS we start with the simplest case: when a link had the same direction of a canonical axis, a link rotation over its axis will be a rotation over the chosen axis (twist rotation). A twist rotation over a link axis can be classified by the hierarchy that has the rotating joint j_k in the segment s_j of a particular vertex v_i (Fig. 4.8).

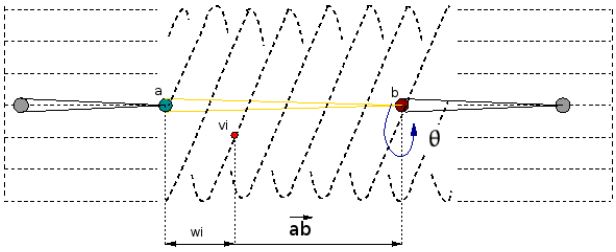


Figure 4.8: *Elements involved in the deformation of the vertex v_i, when a twist rotation θ of the joint b is performed.*

1. One of our segmentation algorithms: (3.2.1, 3.2.2, or 3.2.4) is applied to the target mesh obtaining an additional weight ($w_{\delta i}$); that will be the value of the $\delta(v_i)$ function defined in 3.1, for each vertex and its assigned segment s_j. This value will be stored to be used in step 4.

2. For a vertex v_i in the target mesh, all the joints that influence v_i (the weight w_{ki} for a joint j_k that had influence in v_i is greater than zero) are sorted in a list by their hierarchy (the root joint j_0 has hierarchy 0). Every time a child joint is added, its hierarchy will be increased by one of its father hierarchy value.

3. If a rotation with an angle θ_i is performed over the joint b of the segment assigned to v_i, then θ_i is stored for computation. As we have mentioned, a segment made by the joints: a and b ($\overrightarrow{ba}$), is parallel to the canonical axis in this case.

4. For a joint j_k that is the joint a in the link defined as segment s_j assigned to a vertex v_i. We will compute the rotation angle as: $\theta'_i = \theta_i \delta(v_i)$, therefore if v_i is closest to the joint a of the segment s_j (in this case the joint j_k in hierarchy) the value will be near to 0, or will be θ if its closer to j_{k+1}. The rotation matrix $Mj_{\delta k}$ is computed with θ'_i; in the chain of rotations, $Mj'_{\delta k}$ will be multiplied by Mj_k:

$$Mj'_{\delta k} = (\prod_{i}^{k} Mj_i) Mj_{\delta k} \tag{4.4}$$

5. The joints with lower hierarchy than j_k will need to be rotated in the same angle of the assigned segmentation link axis; then the expression for any joint with a hierarchy lower than j_k will be:

$$M'_{\delta k-n} = (\prod_{0}^{k-n} Mj_i \prod_{k-n+1}^{k} Mj'_h) Mj_{\delta k} \tag{4.5}$$

   Where $\prod_{k}^{k-n+1} Mj'_h$ is an iterative product of rotation matrices; that will had a rotation over the link axis of every joint with higher hierarchy between the joint j_{k-n} and j_k. Therefore a joint with the lowest hierarchy in the chain of rotations; will have its rotation matrix multiplied by all the rotation matrices of all the joints with higher hierarchy in a rotation over the link axis. Having j_k as the joint with the highest hierarchy in the chain of matrix multiplications.

6. Joints with higher hierarchy than j_k will have to be also the same rotation of the assigned segmentation link axis. In a similar way, as the previous point; any joint with higher hierarchy than j_k (j_{k+n}) will need to be multiplied by the negative angle of the link axis of each of

the previous joints. The expression for a joint with higher hierarchy than j_k will be:

$$M'_{\delta k+n} = (\prod_{0}^{k+n} Mj_i \prod_{k+1}^{k+n} Mj'_h) Mj_{\delta k} \tag{4.6}$$

Where $\prod_{k+1}^{k+n} Mj'_h$ is an iterative product of rotation matrices that will had the negative rotation over the link axis of every joint with lower hierarchy between the joint $j_k + 1$ and j_{k+n}.

7. If a rotation over the parent segment of v_i assigned segment is performed; a rotation has been made over the joint j_a of the segment and it has to be taken into account, otherwise the candy-wrapper artifact will be present again (fig 4.9). To prevent this situation, the chain of rotations for influence joints with hierarchy lower than j_a must be multiplied by the rotation chain described in equation 4.5, but with the rotation matrix $Mj_{\delta k}$ as Identity($\theta = 0$).

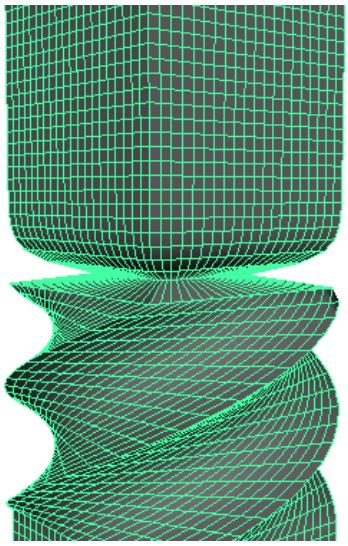
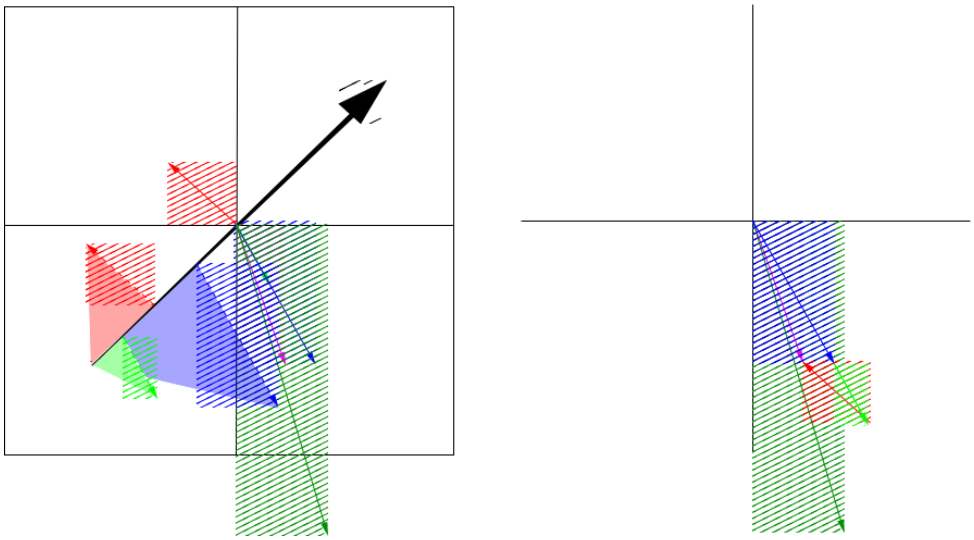


Figure 4.9: *Candy-wrapper artifact in the upper part of a segment produced by not apply our method for joints with hierarchy greater than j_n.*

In the original LBS; if we take the perpendicular plane of the segment link axis and the link as its origin, we can see that the projected vectors from the influenced joints will subtract each other depending of the weights

assigned; producing a new position that moves towards the axis center instead of rotating around it fig. 4.10. As it has been mentioned previously; the idea of avoiding the loss of volume is based on having the same rotation of the segmented link in all the influence joints when a rotation is performed. But only if that rotation it over the segmented link of a vertex. As it can



(a)

Figure 4.10: *Addition of LBS vectors from the influence joints, red, blue and light green. In cyan the resultant vector in the LBS algorithm, in dark green the vector of our approach.*

be seen in 4.5; the rotation expressed by $\prod_{k}^{k-n+1} Mj'_h$ product has to be applied before any rotation from the skeleton space to world coordinates is performed. If the product is applied after the original set of rotation ($\prod_{0}^{k-n} Mj_i$), volume loss artifacts are produced. Putting all the previous cases and information in one expression we obtain:

$$p'_i = \sum w_m M'_{\delta m} M_{Lm} p_i \tag{4.7}$$

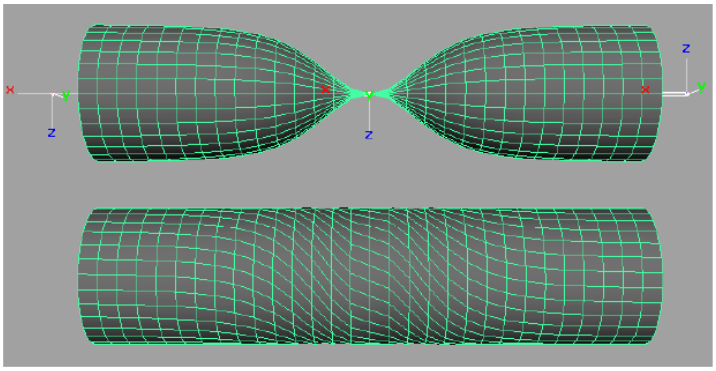
Where $M'_{\delta m}$ will be $Mj'_{\delta k}$, $M'_{\delta k-n}$ or $M'_{\delta k+n}$ depending of the hierarchy of the joint.

Our approach is similar to [50]; but with substantial differences. Our approach has been created with a close relationship with our segmentation algorithms. Since we use the δ function to obtain an additional weight, it has the purpose of being the rotation amount of the total rotation over a link for a vertex assigned to that segment. In [50] they use a similar

parameter ($\delta\alpha_i$) that is based on torsion angles(limited to the X axis) at binding and animation time. ($\delta\alpha_i$) in [50] is computed by $\Delta\alpha_i = \alpha' - \alpha_b$, where $\alpha' = \sum_{i=0}^{n} w_i\alpha_i$, therefore

$$\Delta\alpha_i = \sum_{i=0}^{n} w_i\alpha_i - \alpha_i^b$$

where $\alpha_i = \alpha_i^c - \alpha_i^b$, α_i^c is the torsion angle at a time t of the skeleton animation; and α_i^b is the angle at binding time. Due to this particular way of computing ($\delta\alpha_i$), the output deformation (as can be seen in 4.11) is not as smooth as our method and it may produce some discontinuity artifacts in the vertices that are over the rotated joint. This is a direct consequence of the product with the weights w_i in the computation of $\delta\alpha_i$; where w_i will had its higher values in the joints that are closest to the deformed vertex v_i. Adding to the differences between the method that is used to compute



(a)

Figure 4.11: *The result of* 180°. *rotation over the X axis with the* ***Stretch it*** *deformation method (figure taken from [50])*

the extra weight in both deformation schemes; we believe that the main difference between our method and **Stretch it** [50] is the way the rotation matrix is computed. In [50], the deformation of a vertex is computed by:

$$V_c = \sum_{i=1}^{n} w_i M_{i,c} M_{i,r} M_{i,d}^{-1} V_d \tag{4.8}$$

where

$$M_{i,r} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\Delta\alpha_i) & -\sin(\Delta\alpha_i) & 0 \\ 0 & \sin(\Delta\alpha_i) & \cos(\Delta\alpha_i) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$M_{i,c}$ is the rigid transformation matrix that transforms from world coordinates to local joint j_i coordinate system; and $M_{i,d}^{-1}$ is the rigid transformation matrix that transform from the local coordinates of joint j_i the vertex V_d. As we can see in equation 4.8; the rotation matrix $M_{i,r}$ is restricted to a specific axis (the X axis) and its applied before any rotation is made. The result is similar to our method; but with the difference that the multiplication of the matrix rotation matrix $M_{i,r}$ is applied to any joint that influences the target vertex v_i without distinction. Therefore, the applied extra rotation will introduce an additional rotation in the axis where $M_{i,r}$ is defined, creating an artifact if a precise angle of rotation is required. Another restriction of having the same rotation matrix for every influence joint in v_i; is that the rotation compensation introduced by the matrix $M_{i,r}$ is only correct for the rotated joint j_k. For joints with hierarchy $k+n$ and $k-n$ with $n > 1$ the authors do not specify which is the procedure to apply (as it's documented in our method). Therefore, the compensation of the rotated joint j_k will affect the vertices that are close j_k; and its area of effect will be the links that had j_k as one of its elements. Our approach has a different approximation. Since we apply a different matrix depending on the hierarchy level of the influence joints, we are able to affect only the vertices that are over the link where j_k is the the joint with the highest hierarchy (we use a segmentation algorithm to identify the main link for each vertex), creating a smoother and more natural way of deforming the mesh's vertices. Another important difference between our proposed method and **Stretch it** is that we are not restricted to canonical axis only. In section 4.4 we detail a generalization of our method that allows us to successfully deform limbs that are not aligned with the canonical axis.

General case.

The general case of our approach takes into account the possibility that a limb of a virtual character is not aligned with the canonical axis; therefore when a rotation over a limb is made the rotation had to be over the axis made by link the joints a and b. In equation 4.7 we made the assumption that the link formed by a and b is aligned with one of the axis; in the general case we will take the axis made by the link and we will rotate around the link the segmented vertices of the mesh. The rotation over an arbitrary link can be done with quaternions or with their equivalent in matrix rotation computed by Rodrigues rotation formula given by the next expression:

$$v_R = v\cos\theta + (u \times v)\sin\theta + u(u \cdot v)(1 - \cos\theta) \tag{4.9}$$

if we express the cross product $(u \times v)$ as the rotation matrix $M_{u\times}v$:

$$M_{u\times}v = \begin{bmatrix} 0 & -u_3 & u_2 & 0 \\ u_3 & 0 & -u_1 & 0 \\ -u_2 & u_1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ 0 \end{bmatrix}$$

Rodrigues formula can be expressed as:

$$R_u = I + M_{u\times}^2(1 - \cos\theta) + M_{u\times}\sin\theta$$

therefore $v_R = R_u v$.

When a rotation is applied in a link l_k with an arbitrary position; we use the next procedure to compute the rotation over a vertex.

1. We identify the nearest axis ax_k in orientation with l_k.
2. The angle θ is extracted if a rotation over the ax_k exist.
3. If the angle between ax_k and l_k is greater than 90°; we take the negative value of θ as the rotation angle using the negative of $M_{u\times}$.
4. We apply equations 4.4 or 4.5 depending of the case. But Mj'_k is replaced with R_u with $\theta'_i = \theta_i\delta(v_i)$ as rotation angle; and $\|l_k\|$ as u. Being l_k the link of the segment where the influence joint j_k is assigned.

Volume preservation.

To know how much volume is lost when a rotation is made over a mesh with our rotation scheme; we have made a set of rotations over a mesh. The volume is computed using tetrahedral with negative areas; the result is in the next table with their corresponding set of images. In all cases, the set of weights for the deformation methods are the same. We apply 6 rotations over the joints j_1 to j_3 of the five joints in the bar mesh with an initial volume of 32 units, excluding the end joints (j_0 and j_4) of the test. The set of rotations are planned to show the behavior of every deformation scheme; the results that are shown in table 4.2 are error percentages, where $e_i = \frac{(V_0 - V_i)100}{V_0}$. The rotations in sequence are:

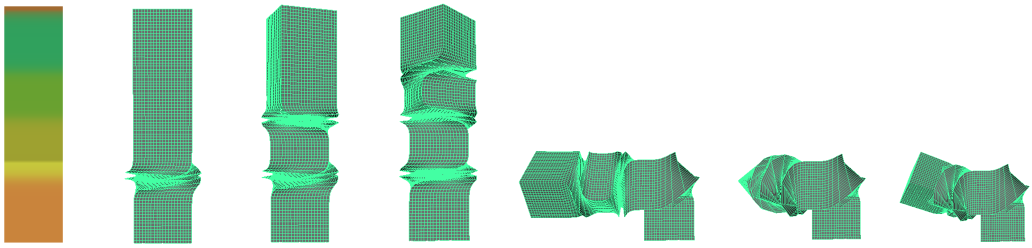
1. 180° in the Y axis, joint j_1.
2. 200° in the Y axis, joint j_2.

3. 120° in the Y axis, joint j_3.
4. 90° in the X axis, joint j_1.
5. 60° in the Z axis, joint j_2.
6. 80° in the Z axis, joint j_3.

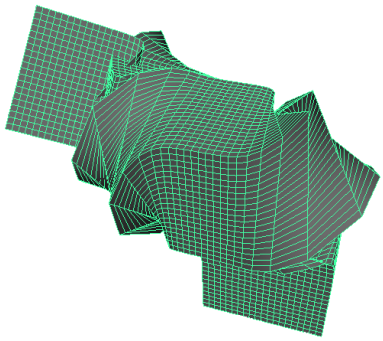
Table 4.2: *Comparative between output volumes from deformation methods (error percentage).*

Rotation #	DualQuat	LBS	P. Method
1	0.0265625%	6.515625%	0.1084375%
2	0.6575%	12.236525%	0.2421875%
3	1.0221875%	16.5721875%	0.3034375%
4	0.9415625%	14 .9853125%	1.9740625%
5	0.6759375%	13.9378125%	2.71875%
6	0.8696875%	13.404375%	3.934375%

In table 4.2 as expected, the method that had lost volume the most is LBS; followed by our method with DQ with the best perform of the three methods. Figures 4.12 and 4.13 shows the surface areas were DQ and LBS operate. The surface areas are each lower than the area were our method operates. This is caused because of the weights values and their behaviour within LBS and DQ. In LBS and DQ the set of weights operates directly over a vertex; but in our method we have two sets of weights. The main (the ones taken directly from LBS) will affect over all the rotations that are not aligned with the segmented link axis. The second one is obtained from the δ function (our $DstF$ and $DtbF$). If a behavior different than the lineal one obtained through the δ function is desired; an output function must be applied over the results δ.
One more difference between our method, LBS, and DQ is its area of action: our method only operates in one segment, LBS and DQ operate in the segments were the weights value is present. This is the main reason of the difference in the affected surface area, but it is also the reason that our method can manage degenerated cases such as rotations equal and greater than 180° because of their progressive nature. As an example of this feature we can see in LBS that if a rotation angle θ is greater than 180°; θ will be equivalent to the difference between θ and 360°. In general, the rotation angle θ in LBS will behave by the relation: $\theta' = \theta - 360(|\frac{\theta}{180}| - |\frac{\theta}{360}|)$; in DQ the rotation about 360° produce serious artifacts as is showed in [20]. Only



(a) Org. Mod. Weights. (b) Rot. 1 (c) Rot. 2 (d) Rot. 3 (e) Rot. 4 (f) Rot. 5 (g) Rot. 6

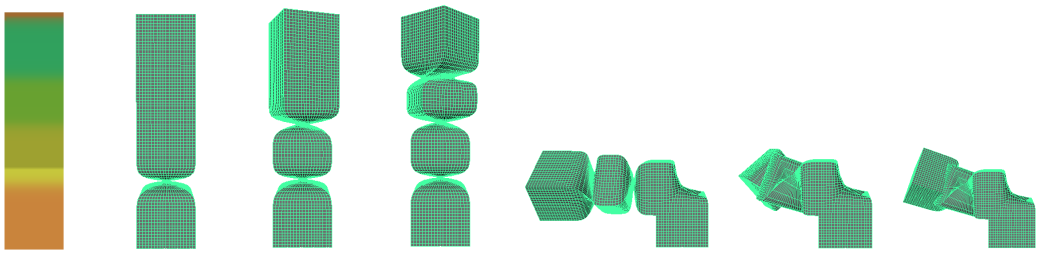


(h) DQ Rot. 6 Perspective.

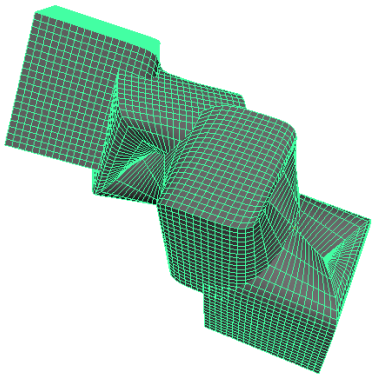
Figure 4.12: *Output volumes from deformation method: Dual Quaternions.*

DIB (Dual Quaternion Iterative Blending) produces a correct output. In our method this degenerated case is properly solved; because θ is changing smoothly between vertex by the δ function instead of changing θ depending on the weights values.
To test how stable is our deforming method, we have modified the bar model. We had made two modifications: one varying down the total volume of our bar and other increasing the volume. The same set of rotations have been applied to this modified models; with the next output data:

As can be seen in 4.3 the variation between the two models are indicative of a stable method. When the results of the set of rotations of the original model (table 4.2) and the result of the output errors on the modified volume models are compared; the output errors are similar.



(a) Org. Mod. Weights. (b) Rot. 1 (c) Rot. 2 (d) Rot. 3 (e) Rot. 4 (f) Rot. 5 (g) Rot. 6

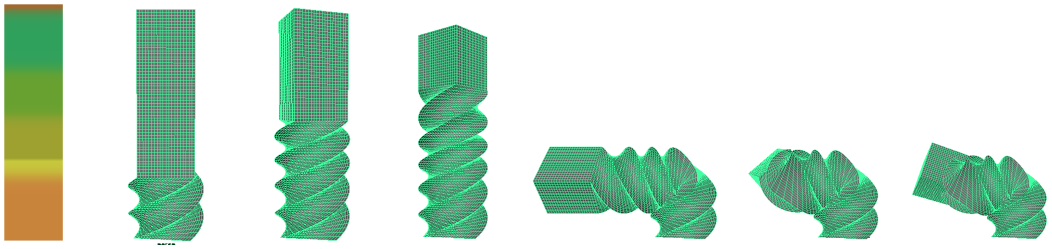


(h) LBS Rot. 6 Perspective.

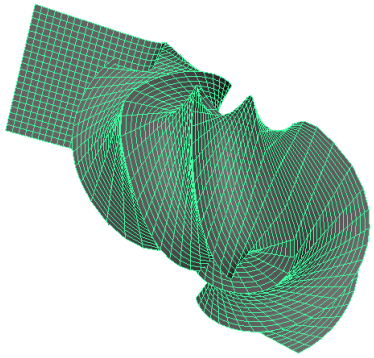
Figure 4.13: *Output volumes from deformation method: Linear Blending Skinning.*

Table 4.3: *Comparative between output errors from two models with different volume magnitude.*

Rotation #	20.8 units Model	72 units Model
1	0.09230765%	0.10833333%
2	0.26634615%	0.24222222%
3	0.34278846%	0.30347222%
4	1.96682692%	1.95611111%
5	2.67355765%	2.71450333%
6	3.89855769%	3.94291667%

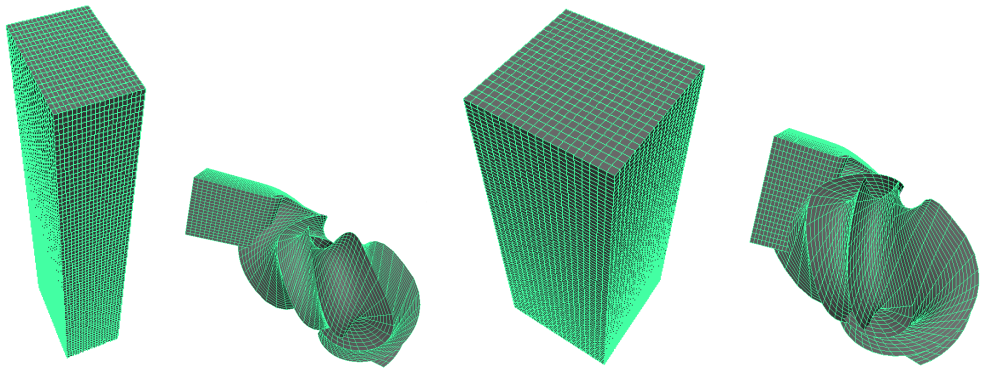


(a) Org. Mod. Weights. (b) Rot. 1 (c) Rot. 2 (d) Rot. 3 (e) Rot. 4 (f) Rot. 5 (g) Rot. 6



(h) Our method; Rot. 6 Perspective.

Figure 4.14: *Output volumes from deformation method: Angular Linear Blending Skinning.*



(a) 20.8 u Mod (b) 20.8 u Mod (c) 72 u Mod (d) 72 u Mod

Figure 4.15: *Modified models comparison.*

Chapter 5

Application in Autodesk Maya.

In the development of the methods described in this document, we have used the Autodesk Maya's API; using $C++$ and MEL languages to implement them as a plug-in. As it's described in the Maya API Guide `http://download.autodesk.com/global/docs/mayasdk2012/en_us/index.html`, we have written our plug-in as a set of commands that are loaded and executed within Autodesk Maya. This commands can be called using the command interface of Autodesk Maya, or can be called defining a custom button in Maya's User Interface that executes the respective command (or commands) with its arguments as a piece of code in the MEL script language.

5.1 Pipeline and commands.

Each part of our pipeline that animates an arbitrary mesh has a specific task associated, and each task or method has a command that performs that specific task. To have control over the pipeline we have developed a Finite-state machine, because each part of our pipeline can be considered as a state. The current state of the mesh is stored as a Maya attribute that is added to the mesh's associated data file in the proprietary Maya file format. The state of the Mesh is saved and loaded inside one of our classes within our Maya plug-in project because it is stored as an attribute.

Skel Command.

The **Skel** command has been made to manage all the tasks that involve the first stages of the creation of a skeleton driven $3D$ character.

vox flag.

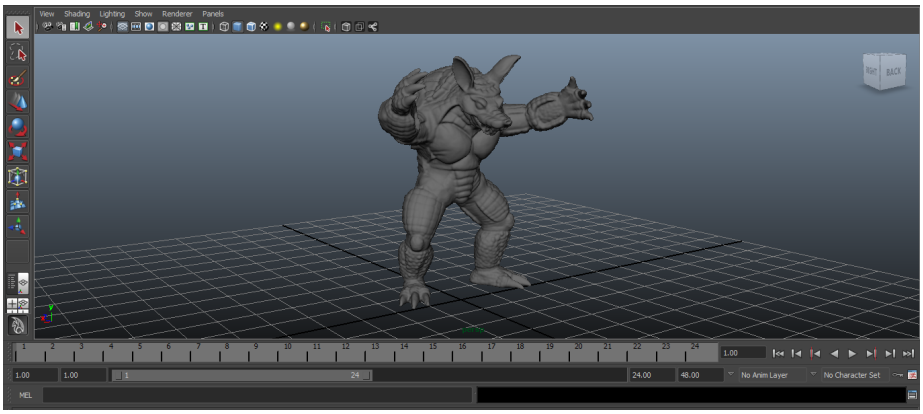
The first stage of our pipeline is to voxelize and fill a closed mesh, as it is explained in section 2.1. The voxelization needs a voxel size, and the voxel size we use is a relation between the size of the longest axis in the bounding box that contains the $3D$ model, and the subdivision number **n**. The **vox** flag voxelizes and fills $3D$ model when the **Skel** command is summoned and the target mesh is selected in the Maya interface. If it's specified, the **sz** flag argument defines the number **n** that is used to compute the voxel size. If **sz** is not specified, the default value of **n** will be 155.
Example: `Skel -vox sz 101` .

thn flag.

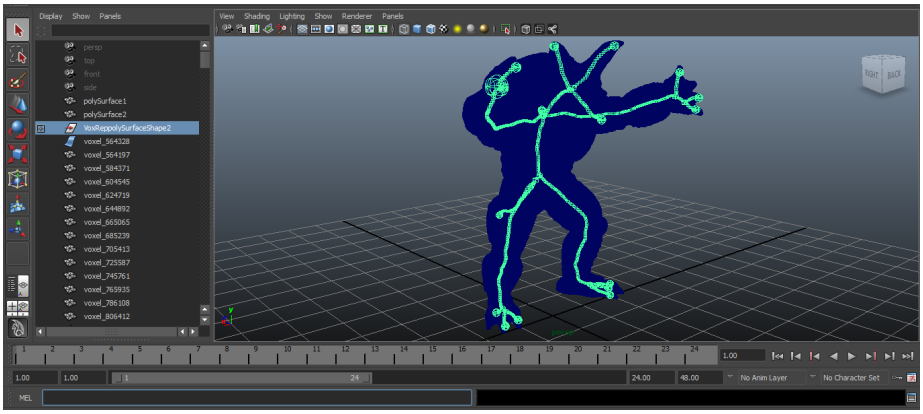
Once the target mesh has been voxelized and filled, a geometric skeleton is obtained through a thinning method (section 2.1); the flag we use in the **Skel** summoning is **thn**, when the target mesh is selected. This flag takes the result of the voxelization process as input, and it only will produce a result if the voxelization was executed previously; Therefore the Finite-state Machine needs to be in a specific state that will be achived only, if the `Skel -vox` command has been applied previously.
Example: `Skel -thn` .

snd flag.

The **snd** flag allows the selection of end nodes (individually or the entire set) for a geometric skeleton; this selection is made by the user. The argument of **snd** is an integer **n** that specifies if all the end nodes are going to be registered in the same action, or if the selection will be made one limb at a time.
The values of **n** are from $[0, 4]$ and -1; where -1 means that the five end nodes are going to be registered at the same time. Starting by the end node that is going to be head, followed by the left hand, right hand, left foot and right foot respectively. If **n** has a different value than -1, only an end node is assigned, 0 for the head, 1 left hand, 2 right hand, 3 left foot and 4 right foot. The **snd** flag needs to be applied after the thinning; the finite-state machine will be change to the correct state until the five limbs are assigned. A user interface window has been made to make easier the end node selection to the user (fig. 5.2).
Example: `Skel -snd -1`.
The **snd** flag applies a geometric skeleton refinement. This means that a pruning over the branches of the geometric skeleton that does not belong



(a) Target Mesh,the Armadillo Monster.



(b) Geometrical Skeleton.

Figure 5.1: *Process of a target mesh in Maya, (b) voxelization and thinning.*

to any of the selected branches is made; also, the root node adjust and a skeleton smoothing (section 2.3) is performed. The result is a refined geometric skeleton such as the one showed in (fig. 5.3) for the case of the Armadillo Monster.

scl flag.

The **scl** flag takes a template skeleton, and based on the length of the geometric segments, the joints are scaled and distributed to the geometric skeleton (section 2.4). During the execution of the **Skel** command with the **scl** flag, an animation skeleton is created (fig.5.4) based on our logic skeleton proxy data.
Example: `Skel -scl`.

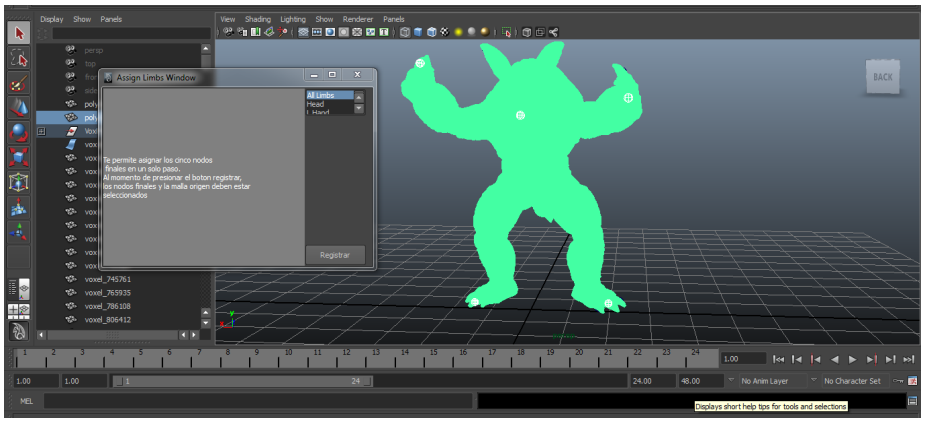


Figure 5.2: *End node selection window*

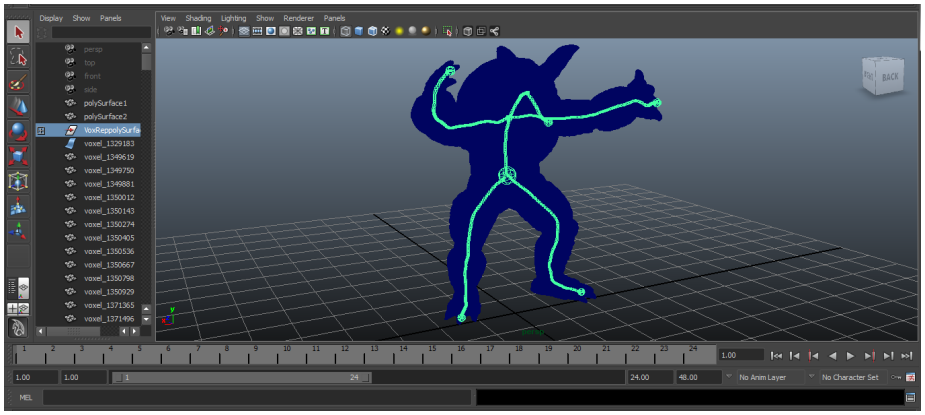


Figure 5.3: *Refined Skeleton in Maya user interface.*

ajM flag.

The **ajM** flag is useful to manually adjust the joints position of the automatic adjusted skeleton; from an assigned node's position to a different one that can be a better choice to avoid artifacts in the animation stage. The flag needs to be invoked with the target joint, and the desired geometric node (represented in Maya by a cube) selected. As it can be seen in fig. 5.5, the position of the target joint will change to the node center.
Example: `Skel -ajM`.

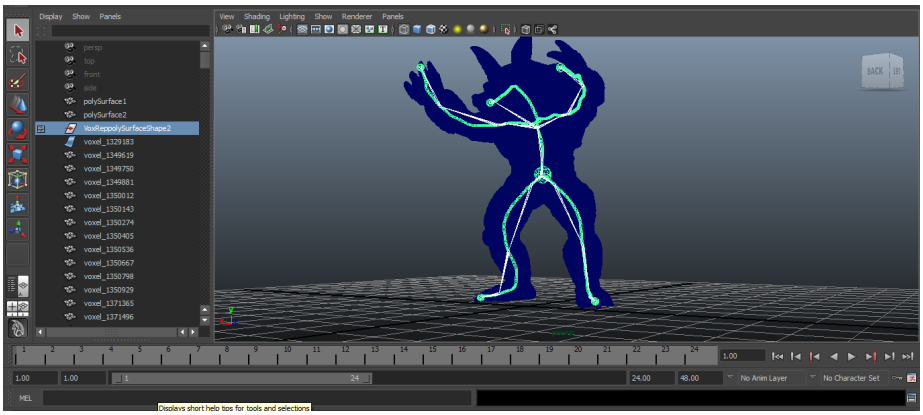


Figure 5.4: *Adjusted animation Skeleton.*

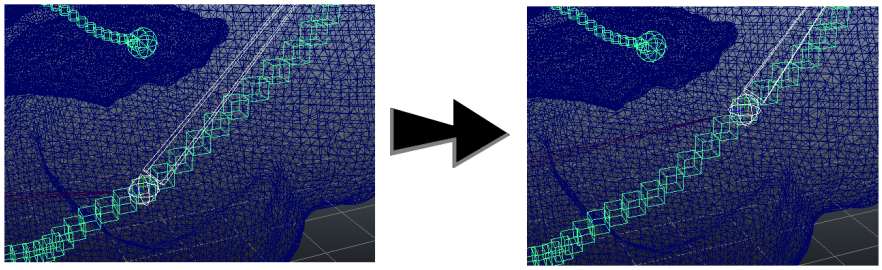


Figure 5.5: *A joint manually adjusted, from one node to another.*

anm flag.

The **anm** flag is a shortcut to the **SmoothBindSkin** command in Maya, with the convenience that only the target mesh needs to be selected, instead of selecting the root of the animation skeleton and the target mesh.

Ajc Command.

The **Ajc** command is created to contain all the methods that are related to animation when the logic skeleton has been created. The application of this command and its set of flags does not involve a finite-state machine to give the user a more flexible environment.

sGt flag.

The **sGt** flag executes the segmentation method 4 on a bound mesh; the flag needs the bound logic skeleton selected to be executed. The argument of the **sGt** is an integer **n** used to define which method of segmentation is going to be applied to the target mesh. If **n** is 0 the simple segmentation method (alg. 3.2.1) is applied, if **n** is 1 the voxelized segmentation method (alg. 3.2.2) is executed, and finally if **n** is 2 algorithm 3.2.4 will be applied; the default value of **n** is 0.
After applying one of the three segmentation methods to the target mesh, the segmentation based weight assigning algorithm (sec. 4.2 is applied to the mesh; the weights of the **LBS** skinning algorithm used in Maya are replaced (using the Maya Skincluster handler) with the set of weights computed by one of the segmentation methods.
Example: `Ajc -sGt 1`.

bSk flag.

The **bSk** flag is used to bind our deformation (skinning) method (sec. 4.4) to a previously bound skeleton in Maya. This restriction is necessary to avoid object inconsistency inside Maya. To implement a new deformation method inside Maya a **deformer** object is developed (deformers are the base class to develop a deformation scheme inside the Maya API [10]).

The deformer needs to be programmed inside a Maya plug-in; loaded with the plug-in and added to the target mesh using the `deformer-type nameDeformer` command in Maya. The **type** flag is used with a string argument that must have the name of the class object of the desired deformer (in the command the deformer's class name is **nameDeformer**). If a smooth bind is applied to a mesh, a **LBS** deformer is created by default; if the **deformer** command is invoked after the smooth binding, the new deformer object will be attached to the output of the Maya's **LBS** deformer. Therefore the output of the target mesh geometry modified by the LBS will be the input to of the new deformer.
The **bSk** flag takes the connections made to the **LBS** deformer as a base to the joint objects inside Maya and reconnect them to the **SkinDeformer** object (the deformer with our deform method), detaching and deleting the Maya's **LBS** deformer, to give the **SkinDeformer** object total control over the target mesh geometry. The **bSk** flag has an integer **n** as the argument; if **n** is 0 a **Skindeformer** will be created and attached to the target mesh.

If **n**'s value is 1, the plug-in command will search in the target's mesh hierarchy of Maya objects if a **SkinDeformer** is attached to a **LBS** deformer, and the unbound and bound of the **SkinDeformer** object process previously described will be executed.
Example: `Ajc -bSk 1`.

anm flag.

The **anm** flag had the purpose of loading an animation file into Maya using the **animCurves** handler. Because of its simplicity and the amount of documentation available, the input format of the animation files we use is the Biovision Hierarchy file format (**BVH**). Inside the command implementation, we have developed a proxy set of classes to represent a logic skeleton, its animation related data and methods. The main advantages of having an internal set of classes are flexibility and autonomy. As an example: if we load a new animation format into our project; we only have to program a class that loads the information of the new format into our internal skeleton set of classes, without needing to change any of the internal functionality of our classes already defined.
The template we use to create a humanoid logic skeleton, is a skeleton with 21 joints that represents the human body, 21 joints does not offer great detail, but we believe it is the minimum number of joints needed to create quality animations.
Animation Skeletons in the **BVH** are defined in the first part of the file, and animation data (animation frames) in the second one. Having a definition part in every **BVH** file has the advantage of defining an arbitrary number of joints per file. At the same time it generates an incompatibility between an already defined animation skeleton and a source animation file, because the animation file can have a logic skeleton with a different number of joints than the animation skeleton. To overcome this problem we programmed the **anm** flag for the **Ajc** command; with the functionality of loading any logic skeleton defined in an animation file and adjusting the animation data (the skeleton's set of rigid transformations that involves its joints), to the joints of our logic skeleton template.
The **Ajc** command used with the flag **anm**, needs the file path of the **BVH** animation file as an argument to be executed.
Example: `Ajc -anm filePath`.

The execution of the command with the defined flag will load an animation into Maya and our project will be ready to be used.

The adjustment between two different logic skeletons needs a more detailed explanation than the one given in the description of the command; due to the relevance of the tool we will give a detailed explanation of the adjust process in the following section.

Adjust between two logic skeletons.

To adjust a source animation skeleton to a target skeleton we borrowed the concept of segments used in section 2.5, and we group them based on their hierarchy level. In this case, a segment is a linked set of joints and these joints are a simple traversal from **joint A** to **joint N**, only **joint A** and **joint N** can have a different number of children than one (cannot be flow nodes). All the joints between the first and end joint will be linked to only one joint (they only have a predecessor joint and a successor joint); and the first and end joint can have more than one or zero children linked, but not one child.
The creation of segments starts from a joint in a logic skeleton (**joint A**); every time a joint with more than one children is found, a new set of segments are created (one segment per child) and the segment that was being created is ended. Segments are classified by their hierarchical level. The concept of hierarchy and segments are closely related; the hierarchy levels are defined every time a joint with more than one child is found during the creation of the segments. The hierarchy value will be increased by one over the hierarchy level of the segment that was used in that moment. Therefore all the segments from a specific level will share the same start element and will end in an element with multiple children as successors (this element will be the start node from a new hierarchy level), or in an element with no children as successors (fig. 5.6). By grouping the segments by their hierarchy, we automatically get a way of creating a correspondence between two skeletons; this is because the joints with more than one child represents specific parts of the human skeleton such as: hips, chest, palms of the hands, etc., and the segments will represent the limbs of a human, the abdomen, the neck, fingers and body parts that are represented by a set of links. Therefore when the segments are grouped by their hierarchy level, it means that we are grouping together body parts and limbs that have the same part of the body as its starting point (fig. 5.6).
To make a correct correspondence between two logic skeleton we have imposed a restriction: the two skeletons need to have the same orientation. This means that the front of the skeletons, the rear, the upper and down needs to be the same, otherwise it could produce artifacts. As an example, imagine that if the front of **skeleton A** is the rear of **skeleton B**; then

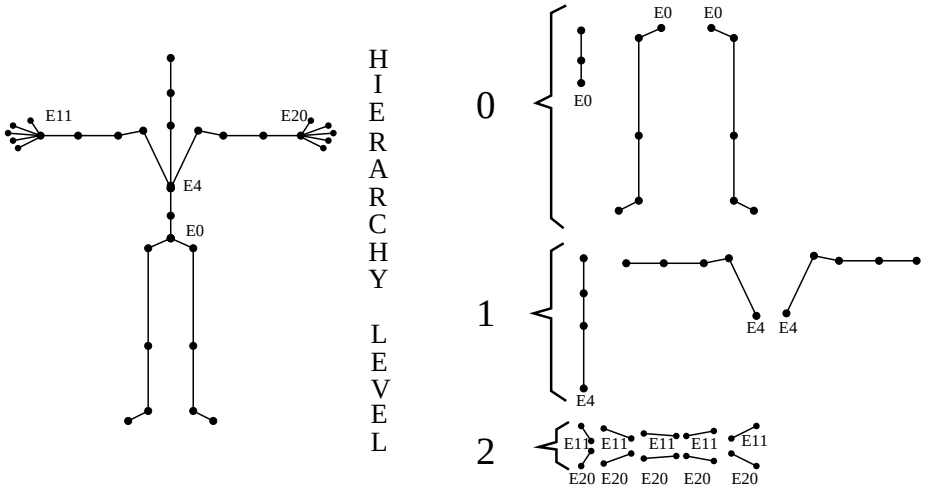


Figure 5.6: *Segmented Logic Skeleton.*

the left leg of **skeleton A** could be assigned to the right leg of **skeleton B** as its corresponding segment. The assignment of the correspondence for two segments of the same hierarchy level is made by a unit vector that has the main orientation of the segment; the segment of **skeleton A** with the same hierarchy level and with the closest orientation to a segment of **skeleton B** will be assigned as its correspondent segment. The correspondence between two skeletons is used to load the animation from a source skeleton (**skeleton B**) to a target skeleton (**skeleton A**).
Once a correspondence between two segments is established, a more refined correspondence is needed, because the number of elements of each segments can be different; as an example: if the segment that represents the left arm of **skeleton A** had 5 elements, and the segment of the left arm of **skeleton B** has 4 elements, the relationship is not direct between the two segments. To solve this problem we compute the length of each segment and normalize it; then a relationship of matching between elements is created. This relationship will be established by assigning the elements with the closest relative length between segments.
If the source segment (the segment from **skeleton B**) has more elements than the target segment (**skeleton A** segment); then the rotation and translation information of the target segment will be a combination of the elements data for the correspondent joint. Therefore, first a map between the elements of the **skeleton B** with the same normalized length of the elements in **skelton A** will be made. Then logically there will be unmapped elements of **skeleton B** to **skelton A**, the unmapped elements will be

solved by combining their transformation information with the next element of **skeleton B**, using its relative length from the previous element to the next element as a factor that multiplies and blend all the intermediate elements between a mapped element in **skelton B** and **skeleton A**. As an example, from the figure 5.7, we can see that the element of **skeleton B** with a length equal to 0.583 maps to an element in **skeleton A**, but the element mapped in **skeleton A** will have combined the information from two elements in **skeleton B**: the mapped and its predecessor, with a factor of 0.6 for the element that has the correspondence and 0.4 from the predecessor.
On the contrary if the source segments have fewer elements than the target segment; the elements of the source segment will be assigned to the element with the closest normalized length in the target segment. The elements in the target segment that has not assigned a correspondence element will be loaded with information equal to zero (zero rotation and translation). If the two segments have the same number of elements a direct relationship will be created. The result of this adjustment between logic skeletons is a method

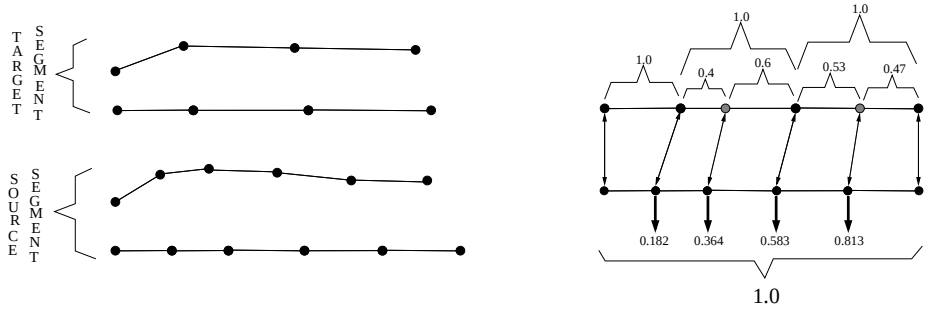
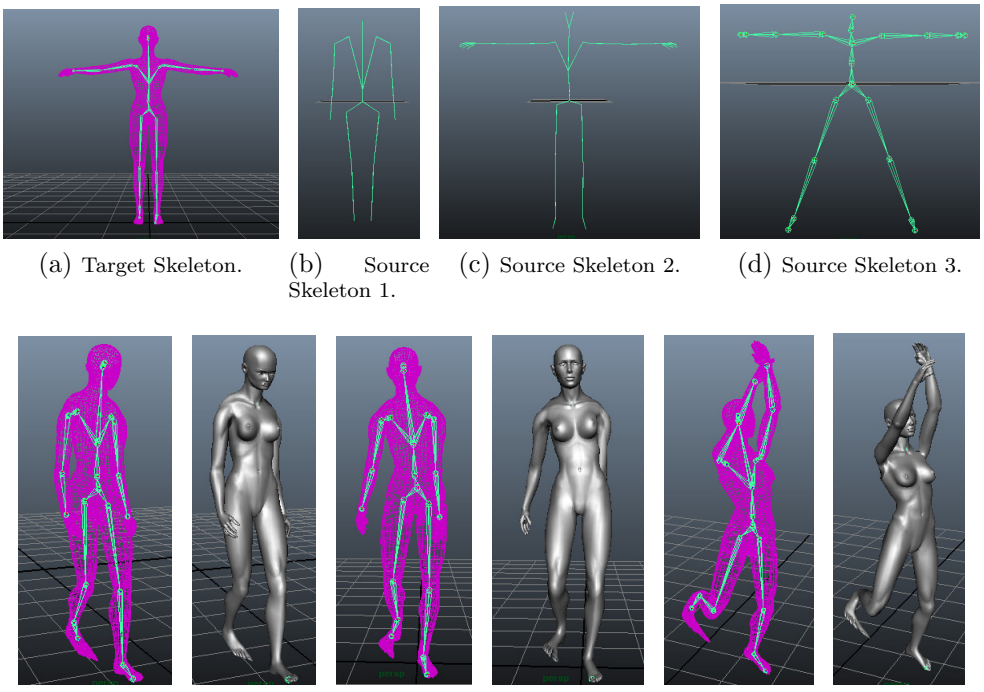


Figure 5.7: *Correspondence between segments.*

that is useful to load any **BVH** file without restriction of its number of joints, that has the right orientation into our generated logic skeleton. As it's shown in fig. 5.8, not all the logic skeletons can have the same initial pose defined. To solve this problem, we computed a set of rotations from the initial pose of our target skeleton to the source skeleton's initial pose. We call this set of rotations over the skeleton: a rotation offset, and is added to all the frames loaded into our target skeleton in Maya.
Once the offset is added to the set of translations and rotations of the target skeleton, the results in an animated character are appropriated; but are also dependent of the position of the joints within the target mesh. If joints' positions of the target skeleton are refined by a human user, the results are better than the ones that are taken from the automatic ones generated from

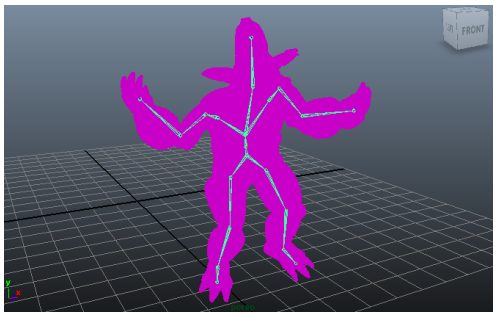
the **Skel -scl** command; as an example in fig 5.9 (a) we show the effects of a bad positioned joint within the armadillo monster mesh (the head is assigned to the nose and an animation artifact in the target mesh is created because of this bad choice); fig. 5.9 (c) shows the same animation with a correct head joint position.
A correct correspondence between logic skeletons is dependent of the pose of the target skeleton. The closest that the target skeleton pose is to the source skeleton pose; the lower than an artifact will appear in the loaded animation.



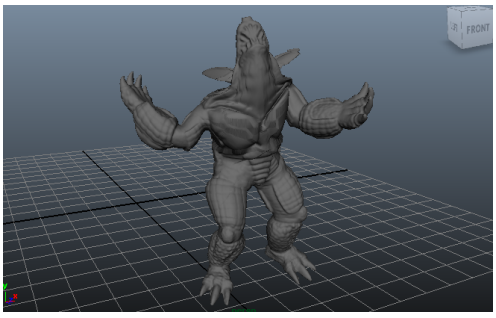
(a) Target Skeleton. (b) Source Skeleton 1. (c) Source Skeleton 2. (d) Source Skeleton 3.

(e) Trgt. Skel. wrfrm. (f) Trgt. Skel. 1. (g) Trgt. Skel. 2 wrfrm. (h) Trgt. Skel. 2. (i) Trgt. Skel. wrfrm. (j) Trgt. Skel. 3.

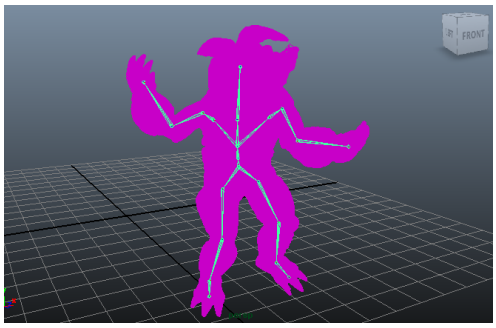
Figure 5.8: *Target Skeleton (a), and different Source skeletons (c-d), the animations of each source skeleton adjusted to the target skeleton (e-j).*



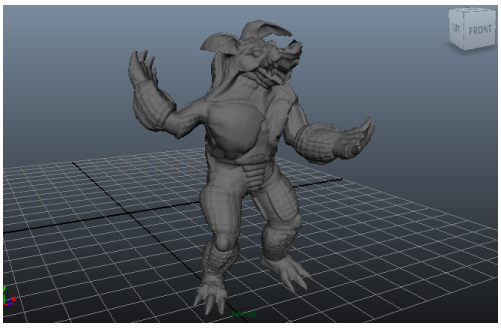
(a) Target Skeleton with artifact (wireframe).



(b) Target Skeleton with artifact (shaded).



(c) Target Skeleton without artifact (wireframe).



(d) Target Skeleton without artifact (shaded).

Figure 5.9: *Effects of the positions of the joints in the target skeleton when loading an animation file.*

Chapter 6

Results, Future Work, and Conclusions.

The main objective of this thesis was to create a method to animate a character mesh without an animation rig, skeleton, or animation technique associated. Our method has the intention of being a tool that can be used to automatize or speed up the process made by a digital artist. Our target users are people without previous experience or with minimum knowledge about the topic, and experienced users who wants to save time on the rigging process. We believe that we have succeed in general terms in this main objective, but with certain limitations. Our method is restricted to human-like closed meshes due to the difficult task that is rigging any arbitrary mesh.

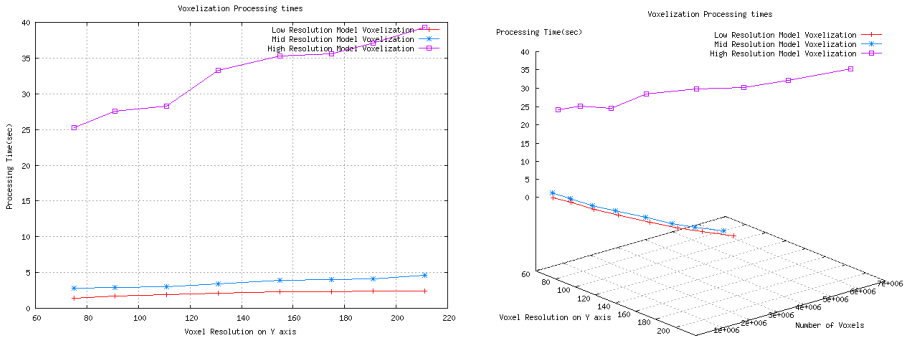
In this chapter we are going to show the results of each of the elements involved in every stage of the proposed methods to create an animation rig; a discussion about the drawbacks and advantages of the selected and developed methods, a comparative with existing methods, and finally the future work and the lines of research to improve and extend the capabilities of the presented algorithms and methods.

As it has been mentioned in chapter 5, all the developed algorithms are single thread and were coded in C++ as an Autodesk Maya ©plug-in; the method's invocation are through a Maya command within the Maya framework. The Autodesk Maya version used in the development of this thesis has ranged from the 8.5 64-bit up to Autodesk Maya 2013 64-bit edition. The Operating systems used was Windows XP-64 bit with Visual Studio 2008 C++ compiler and a Windows 7-64 bit with a Visual Studio 2010 C++ compiler. All the plug-in versions have been coded with the 64-bit platform option. The hardware used in all the test and screen shots was an Intel COREi3-2310M at 2.10 GHz with 6 GB of DDR3 RAM and a

NVIDIA Geforce GT-540M with 1 GB of dedicated video memory.

6.1 Voxelization, thinning, and Skeleton adjust algorithms.

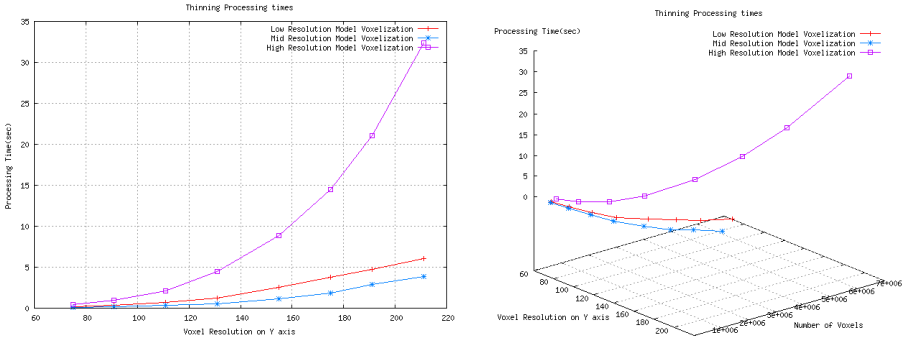
Voxelization and thinning algorithms are some of the most time consuming part of a character rigging within our pipeline. As we had mentioned in section 2.6, we have used as default voxel size an approximated of 0.65% of the model's height (we define the default value as the variable $n_V = 155$ voxels length of the longest axis in the bounding box that surrounds the target model mesh). The default voxel size was chosen based on tests with different voxels sizes applied to three different meshes with: low$(6,488)$, medium$(15,576)$ and high$(172,974)$ number of vertices respectively; 0.65% was the size that give us the best results in terms of processing time and space resolution to extract a geometric skeleton through the thinning process, this is a common way of choosing the voxel size for methods that work whit voxels [8]; due to the complexity to propose a clear set of rules to determine the optimal voxel size that depends on the number of vertices and other important features for the applied method. Table 6.1, graphs 6.1, 6.2, and 6.3 shows the results from the voxelization and thinning processing times with different voxel resolutions; with n_V ranging from 75 to 211.



(a) Processing times *VS* voxel resolution. (b) Processing times *VS* voxel resolution *VS* voxel number.

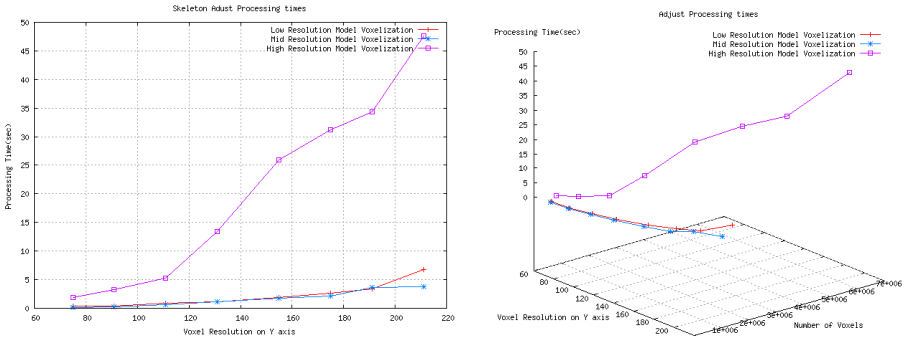
Figure 6.1: *Processing times of voxelization.*

As is seen in graph 6.1, the processing times of the voxelization increase along with the resolution in voxels, but the slope is not as pronounced as the slope in the thinning process. The reason is that the voxelization



(a) Processing times *VS* voxel resolution (b) Processing times *VS* voxel resolution n_V *VS* voxel number.

Figure 6.2: *Processing times of thinning process.*



(a) Processing times *VS* voxel resolution. (b) Processing times *VS* voxel resolution *VS* voxel number.

Figure 6.3: *Processing times of Skeleton Adjust.*

algorithm depends more on the vertex number than in the voxel resolution. In the case of the thinning and the adjustment, they are dependent on the voxel number due to the thinning algorithm nature of deleting voxels iteratively from the surface of the voxelized model. The adjusting algorithm creates voxel segments and traverse them to adjust the logic skeleton to the extracted geometric skeleton; therefore, the processing times will grow along with the number of voxels.

Model	*Vxlztn.(sec.)*	*Thn.(sec.)*	*Vxl. Num*	*Vxl. Res.*	*Adj.(sec).*
Low res.	1.4	0.18	77700	75	0.38
Low res.	1.73	0.33	139230	91	0.43
Low res.	1.9	0.69	256410	111	0.86
Low res.	2.06	1.23	422475	131	1.14
Low res.	2.31	2.58	687735	155	1.91
Low res.	2.34	3.81	999075	175	2.64
Low res.	2.38	4.71	1292688	191	3.37
Low res.	2.45	6.02	1755520	211	6.82
Mid res.	2.77	0.08	61200	75	0.18
Mid res.	2.94	0.17	113295	91	0.27
Mid res.	2.99	0.34	201798	111	0.62
Mid res.	3.45	0.57	327369	131	1.08
Mid res.	3.87	1.17	542500	155	1.75
Mid res.	3.98	1.87	779100	175	2.14
Mid res.	4.07	2.87	1024333	191	3.62
Mid res.	4.65	3.9	1370234	211	3.74
High res.	25.28	0.47	274050	75	1.83
High res.	27.54	0.95	490490	91	3.31
High res.	28.25	2.12	886890	111	5.32
High res.	33.28	4.48	1441000	131	13.42
High res.	35.28	8.84	2416295	155	25.94
High res.	35.54	14.49	3447150	175	31.23
High res.	37.11	21.01	4489646	191	34.28
High res.	39.25	32.39	6046838	211	47.64

Table 6.1: *Processing times.*

Comparative.

The main methods for comparison are the works developed by Baran and Popovic [3], Pan and Xiaosong [34], and Bharaj and Thormählen [4]. In all these works, the skeleton extraction is different from the voxelization and thinning used in our work. In [3], instead of extracting a geometric skeleton and adjusting a logic animation skeleton; they embeded an animation skeleton to the target mesh. The embedding of a skeleton is an interesting idea that allows to animate a character with an approximate pose to the skeleton; but with low resemblance to the shape that is going to be animated by a skeleton. However we believe that this is a plus feature and is not a common case when someone is animating a character mesh since the common case is to create a skeleton that has the same general shape and topology information of the target mesh. In [4] the input can be a multi-component mesh (a character made by multiple meshes), and the skeleton is extracted by computing a contact graph for each mesh; the skeleton is generated by joining each contact graph with a clustering algorithm. We believe that this is an important feature; because it allows the use of a wider variety of input $3D$ models than our method and any other methods mentioned here. Finally in [34], they produce a curve skeleton from a single input mesh; using two 3D *silouettes* (a 2D projection with its **Z** coordinate) by computing

the mid points of the internal edges of the Delaunay triangulation over the main silhouette. The z axis is refined with the second silhouette.
Methods depicted in [3] and [4] automatically adjust and rig an animation skeleton to a character mesh; method [34] only produces a rig based on the input mesh geometry. Therefore its output lacks the practical application that our method and methods [3] and [4] had. Our method, on the contrary, needs the assistance of a human user; we have taken this strategy because it allows us to apply it to a wider range of input $3D$ models than [34] and [4]. In both compared methods after extracting a curved skeleton (or skeletons in [4]); the main problem lies in finding the nodes that had a correspondence with the animation skeleton joints from the extracted skeleton. In [34], a minimization method is applied to the end nodes of a reduced skeleton. In a similar fashion, in [4] the final nodes of the reduced skeleton are detected to feed a minimization problem; to find a mapping between the high resolution graph (their version of an extracted skeleton) and the animation skeleton. In [4] the root node is found by applying an algorithm to find the betweenness-centrality of the internal nodes in a graph. In [34] the root node is founded implicit by the penalty functions used to embed the animation skeleton to a reduced skeleton. In both works [34] and [4], one of its main limitations is that the input meshes must have a similar pose as the input animation skeleton to work in a proper way. We have proposed a method that, as the mentioned works, use the position of the end nodes to automatically map the animation skeleton end joints to the end nodes of the extracted skeleton [36]. For that reason we decided to leave the end joints labeling task to a human user to allow him to apply the skeleton adjust tool to any closed human-like mesh in an arbitrary pose. Whit an automatic rigging tool, another problem raises: the animation captured or designed for the original mesh will not be reproduced with the same poses in the target meshes; if the target meshes have little resemblance in their pose with the initial pose of the original animation skeleton. In papers [34] and [4] this problem was not addressed. In the developing of this thesis we have designed an animation transfer tool 5.1 that creates correspondences between an animation skeleton and an arbitrary animation file with a human-like animation skeleton. One of the problems that we faced was the difference between initial poses. We had solved this problem by adding an extra animation frame to the loaded animation to correct the initial pose from the extracted skeleton to the loaded animation file.

Future work.

As future work we would like to develop a skeletization algorithm that can work with multiple input meshes, without the restriction of having a closed mesh as an input as in [4]. The voxelization is a powerful tool, but is restrictive and depending on the voxel resolution it can have high processing times. A tool based solely in the mesh vertices is desirable. Another natural step to improve our work will be designing an algorithm that identifies the end nodes in the extracted skeleton of an input mesh with arbitrary pose and orientation (not restricted such as the [3] and [4]); and find their correspondence for an animation skeleton. We believe that this problem is in particular challenging; and a novel way to find an alternative to solve it will be through a geodesic mapping of both skeletons (in a similar way as the one used in [19]) producing an invariant pose. Having defined this invariant pose we will find the correspondences between the nodes from the extracted skeleton to the animation skeleton; even for non *human-like* meshes using a set of template skeletons. A way to improve our skeleton adjustment tool would be implementing a more effective method to place **grade two** joints (such as elbows and knees) of the adjusted skeleton; whenever the shape and pose of the input mesh allow it.

6.2 Segmentation, Weight Generation, and Skinning algorithms.

The segmentation algorithm is the core concept we use to generate the weights for the *LBS*, and is also used in our skinning algorithm; due to its close relationship with the two algorithms we decided to put the results of each one in the same section. As it can be seen in table 6.2, the times of our segmentation algorithms depend on the number of vertices of the input model mesh. In the case of the Voxelized Segmentation algorithm, when the number of voxels is too high as in the case of the High resolution model, our equipment was unable to complete the task due to a memory limit problem (the quantity of memory demanded was too high). This is because a test is applied to every vertex of the input mesh model; using a test to check if the lines that joins the vertex with each one of the joints are inside or outside of the model. Our region-grow based algorithm (algorithm 3.2.4) has the best processing times of the three, it is also the most flexible and has good results with characters in *T-Pose* and with arbitrary poses as well; their processing times are from 7.30 to 1.4 times faster than algorithm 3.2.1 restricted to *T-Poses*; and it's about 106.34 times faster than algorithm

3.2.2, which uses voxelization to manage arbitrary poses. Algorithm 3.2.4 also has the best results in terms of segmentation quality, surpassing the results obtained with algorithm 3.2.2.

The weight assignment algorithm uses segmented vertices; therefore the segmentation method used on the input mesh does not have an impact on their processing times (the processing time depends of the number of vertices in the input mesh). But their output values depend on the segmentation output. As we mentioned in 4.3, the values of the weight assignment algorithm are the result of the distribution function used along with algorithm 4.2.1; any function can be used in our algorithm because the weight assignment algorithm focuses in the joint hierarchy assigned by the segmentation algorithm. Therefore, the chosen *distance* and *distribution* functions depend on the desired effect or behavior that the artist wants to apply to the segmented input mesh.

Model	*Num. Vert.*	*Seg.(sec.)*	*Vxl. Seg.(sec.)*	*Reg-Grow. Seg.(sec.)*	*Wght. Assg.(sec.)*
Low res.	6,488	2.85	41.53	0.39	16.53
Mid res.	15,576	4.41	93.46	0.88	21.45
High res.	172,974	45.49	-	32.3	293.51

Table 6.2: *Segmentation processing times.*

Is customary in the skinning subject to make a comparative of the performance of a new proposed skinning algorithm with the most popular algorithm due to its simplicity and its linear nature: **LBS**; *LBS* has the best performance of all the skinning algorithms used up to date. We also compared the performance of our algorithm with another popular skinning solution: **DQS**. *DQS* has a lower performance than *LBS*; but it solves one of its main problems: the well-known candy wrapper artifact. Therefore this are the two main algorithms to compare with a new proposed method in the field. As we have done in section 4.4; we applied a sequence of six deformations in a test model (a bar) that are reported in table 6.3. As it can be seen, our method without any optimization is about six times slower in the worst case compared with *LBS* and three times in the best case. When compared with *DQ*, our method performance is four times slower in the worst case and two times in the best one. The slower performance of our method is caused by the extra matrix multiplication on the influence joints of each vertex; according to the hierarchy of the influence joint, is the number of extra Matrix multiplication that are needed to be done. Another situation that affects the performance of our skinning method is: for each vertex, the joints need to be sorted by hierarchy. We use a Quicksort

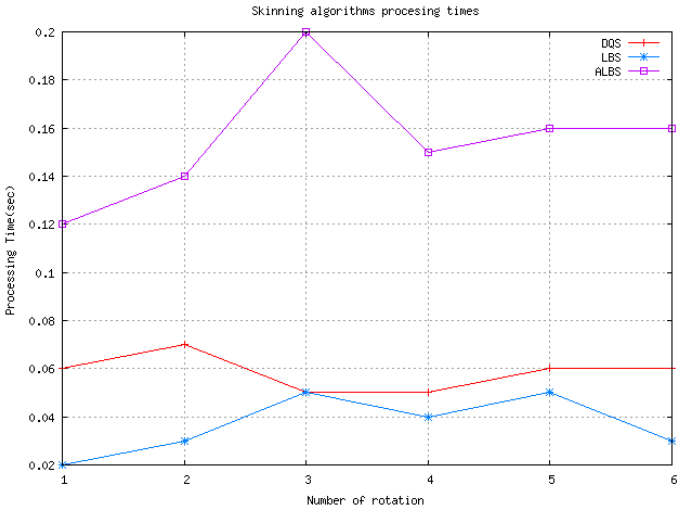
base 2 algorithm to solve this problem in the most optimum way; but the extra computation is reflected in the final processing times. Although the *LBS* and *DQ* algorithms are faster than our algorithm in performance; our algorithm shows proper results without the candy wrapper artifact of the *LBS* or the artifacts caused by the weight distribution showed in the *DQ* algorithm. To improve the performance of our Segmentation Based Linear Blend Skinning method (*SLBS*), we have decided to make an optimization; our main idea to improve our computation performance was to restrict the influence joints for each vertex to a maximum of four. Four joints per vertex is enough to have quality deformations, and a common practice among the animation of human-like characters. Another optimization made to our code was to precompute the matrix chain for all the vertex in a segment, leaving only to compute the twist rotation matrix specific for each vertex. With the optimization, *SLBS* has a great improvement in its processing times, having a better performance than *DQ* and being almost as fast as *LBS*, but with a superior deform quality than the two compared methods.

Table 6.3: *Comparative between deformation methods (Processing time).*

Rotation #	DQ	LBS	SLBS	op. SLBS
1	0.06 ms	0.05 ms	0.12 ms	0.05 ms
2	0.07 ms	0.04 ms	0.14 ms	0.05 ms
3	0.05 ms	0.05 ms	0.20 ms	0.05 ms
4	0.05 ms	0.04 ms	0.15 ms	0.05 ms
5	0.06 ms	0.05 ms	0.16 ms	0.05 ms
6	0.06 ms	0.04 ms	0.16 ms	0.05 ms

Comparative.

The segmentation in the automatic rigging algorithms [4] and [34] are a direct consequence of their respective skeletonization methods. In [4] the segmentation is called rigid skinning; the value of the weights of a vertex for all joints will be 0 with exception of the assigned joint that will be 1. The segmentation in this case is obtained by automatically collapsing the vertices in the high resolution graph of the skeletonization process. The method proposed in [34] segments the vertices of the target mesh in one of the steps of the skeleton generation; using a distance function from the set of vertices to the branches (their equivalent to our segments) of a $2D$ silhouette for each vertex on the target mesh.



(a) Processing times of Skinning algorithms.

Figure 6.4: *Comparative of processing times of Skinning algorithms.*

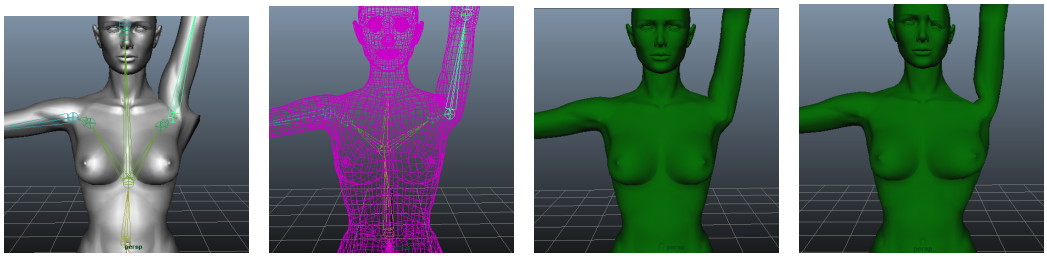
The methods previously mentioned, share one main feature in their segmentation: the segmentation is part of the skeleton extraction process; therefore, their segmentation will fit perfectly with the segments of their output skeleton. Our case is different: we are not working directly with the vertices of the target mesh in the skeleton extraction stage; instead we work with a voxelized version of the input $3D$ mesh, and in the thinning process we lose track of voxel equivalences with the input mesh vertices; therefore we were in the need for an algorithm that works with a rigged mesh as an input. If the segmentation algorithm is not dependent of a skeletonization algorithm, it can be applied to any rigged input mesh. The segmentation can be useful as a guide to digital artist to known which is the main influence joint for every vertex in the input mesh. The disadvantage is that the difficult is greater than the segmentation made by an algorithm that works with the vertices positions to extract a skeleton. The main problem in a segmentation algorithm that uses a rigged mesh as input is knowing which is the main influence joint for every vertex; specially if the input mesh is not in the ideal *T-pose*. When a model is in *T-pose*, the segmentation problem is easier; there is no overlapping of any of the limbs in a $2D$ projection of the mesh, and the closest link segment to a vertex is the correct choice. In an input mesh with an arbitrary pose that is not the case; the problem relies in knowing which is the closes link in the skeleton to a particular vertex with a valid traverse within the target mesh volume.

Algorithms like [14] and [6] uses as input a set of meshes; with the main inconvenience that most of the $3D$ modeling in films and video-games are created as a static input mesh, and then animated. Both of this methods use segmentation to extract an animation skeleton that fits the animation of its input mesh. We instead adjusted an animation skeleton to an existing one from a static mesh. The method used in [39] is similar to one of our segmentation algorithms; they also uses a voxelized version of the input model. But their distance measure method is based on geodesic distances instead of using Euclidean distances; they also use their segmentation algorithm as the base to a weight assignment algorithm such as the one presented in [3]. Their segmentation algorithm seem to be effective with models with a large non-convexities, but they apply it only to quadrupeds models in a "neutral pose" (the pose equivalent to a *T-pose* in human-like mesh models) and does not shown if it is effective in arbitrary poses.

Our automatic weight assignment algorithm was developed with the same objective as the one showed in [3] that is: to create a set of weights having only a $3D$ mesh as an input. Other algorithms such as [24], [46], [28], [29], [27], [47], [22] and [12], had a set of examples to compute (or extend in some cases) the weights of each vertex in a character $3D$ mesh. As it's noted in [39], a good segmentation is a good starting point for a weight assignment algorithm, such as [4] and [39]; in these works they use their segmentation (rigid skinning) as a base for the heat diffusion automatic weight assign algorithm proposed in [3]. In [49], an alternative based on a high order function of an influence ratio r is depicted; which is dependent of the neighbors joints and a *user defined* parameter α. In this work the α parameter is used to work properly without a segmentation algorithm. Their weight distribution is based on three rules such as *symmetry*, *smoothness* within and in the boundaries of the influenced region; such features are important and are present in the heat distribution algorithm described in [3] and in the *Gaussian* functions used in our *distribution function*. So we can say that the main features that an automatic weight algorithm needs to work properly are a good segmentation and a distribution function that fulfilling the three features previously mentioned. Our developed algorithm has all those three features, but this features are a consequence of the chosen *distribution function*.
The core of the weight assignment algorithm is the segmentation that not only creates a rigid skinning weight distribution (such as [39]) it also stores information about the segmentation (the main influence link per vertex and neighborhood information of the skeleton); allowing us to create an easier and straight forward algorithm to assign weights to a LBS based deforma-

tion scheme. Another key piece to improve the output of the weight assign algorithm is a correct joint placement. In [22] an elastic energy-minimizer is used to improve the weights obtained with *Bounded Biharmonic Weights (BBW)* [13]. In our case some artifacts can be solved by a correct placement of the skeleton joints (manual placement) as can be seen in fig. 6.5.

Our developed skinning method is an extension of the *LBS* algorithm;



(a) Incorrect joint placement.

(b) Correct joint placement.

(c) Correct joint placement(shaded).

(d) Autodesk Maya default automatic weight assign output.

Figure 6.5: *Artifacts generated by a incorrect joint placement using our weight assign algorithm, figure* ***d*** *artifact induced by the Maya default weight assign algorithm.*

which works on rotations over the links axis of the input mesh model to overcome the well-known *candy wrapper* artifact. As we mentioned at the end of section 4.4, our method is similar to the one published by Yang [50] with their respective differences (differences we described in section 4.4). Another main point of comparison in the case of the skinning algorithm is the work developed by Kavan in [21]; *DQS* uses the same weight influence base of *LBS* and also corrects the *candy wrapper* artifact by using non-linear trajectories in the deformations over the link axis, but it also introduces bulging artifact in rotation over limbs such as elbows or knees by the non-linear (similar to a sphere) of *DQS*. Another problem addressed in [20] is the artifacts that *DQS* creates when the rotation over the link axis is close to 360°. Artifacts that our solution does not have with the tradeoff of having longer computing times than *LBS* and *DQS* in the non-optimized version of our algorithm, but similar in quality to *Dual Quaternion Iterative Blending (DIB)* which is more than five times slower than *DQS* [20], when our method is approximately from two to three times slower, and four times slower in the worst case; and in the optimized version is $0.01ms$ slower than *LBS* and faster than *DQ*. The most recent and sophisticated algorithms are [20], [12], and [22]; all three cited algorithms had something in common:

they are an extension of *LBS* and use additional weights to improve the *LBS* behavior.

In [20] a set of additional bones along with their respectively weights are used to mimic the behavior of nonlinear deformer, such as *DQS* or *DIB*; the technique adds virtual bones to a skeleton (which are controlled by a β_{max} parameter defined by the user) by an optimization problem where the energy error is minimized and the original and extra weights are recomputed. This method is a fast one, because it's essentially *LBS* and the overhead in time and memory is caused by the extra bones added to the skeletal structure; therefore it's faster than our non-optimized method but at the price that the desired behavior needs to be approximated with a set of samples in the precomputed stage. Also, an analysis of the derivatives of the non-linear method that wants to be approximated must be done; and the well-known artifacts of the *LBS* may be present for cases that were not being considered as an input while in our method no additional analysis or input frames are needed. The method developed in [12] is similar to [20] because it uses abstract handles (a concept that close resembles virtual bones) to improve the *LBS* deformer. The abstract handles came along with their set of weights that are appended to the rest-pose matrix of the *LBS* algorithm. Abstract handles are spread in a region automatically by a multi-dimensional farthest point method in some vertices of the input mesh. The result of this method improve greatly the quality of the *LBS* when dealing with the usual loss of volume and their performance is good; but is not clear whether if this method can remove properly the candy wrapper artifact or how it behaves with twist rotations higher than 360° a feature that it is properly solved by our method. The method used in [22] is similar to our approach in the sense that they use *LBS* as the base for rotations in local coordinates XY planes (swing rotation); for rotations over local z axis (twist rotation) they change to a nonlinear interpolation method (an approximation to *DQS*). They also apply the twist rotation in the middle of the link segment and not over the target joint; in a similar way we apply our extension to *LBS* when a rotation is made over a segment link axis. Their performance is not clear due to its lack of times on the skinning stage; their main overhead is present when the two deformers are evaluated. Because of its close relation to *DQS*, we believe that deformations closer or higher than 360° can lead to artifacts present also in [20] that are properly solved by *DIB*, we created the table 6.4 that shows the features that our method share with other geometry based skinning methods for a better understanding and comparison of their main advantages.

Features	DQS [21]	SLBS	LBS	DIB [21]	ALNS [20]
Uses LBS weights.	√	×	√	√	×
Solves candy wrapper artifact.	√	√	×	√	√
Solves or not produce bulging artifact.	×	√	√	×	√
Prod. correct res. with rot. over 360°.	×	√	×	√	?
Short processing times.	√	√	√	×	√

Table 6.4: *Skinning algorithms main advantages comparison.*

Future Work.

Our method is far for being perfect and can be improved in each of its stages. The segmentation stage would be easier if the input rigged mesh was in the ideal **T-Pose**; if an arbitrary mesh is downloaded from Internet or some other source there is a high possibility that the selected mesh is in an arbitrary pose. To solve this problem we would like to explore an algorithm that uses geodesic distances (such as [19]) to unfold the arbitrary pose to something closer to **T-Pose**; and then appling our simple but reliable segmentation algorithm that is based solely in the closest Euclidean distance between a vertex and a segment instead, of a more restrictive, demanding and complicated algorithm like algorithm 3.2.2 based on a voxelized mesh. Another way will be to improve algorithm 3.2.4 using a different way to measure distances than Euclidean distances, we would like to apply geodesic distances as the method to measure distances. We believe that the region merge post-process needs improvement, because in some cases it can assign areas in an improper way if the data in the model (number of vertices) is low.

To improve our skinning weight assignment algorithm we would like to explore an algorithm based on examples; such as [22] that allows us to apply the information obtained by examples in models with similar shape. We use a Gaussian function as weight distribution function; but we want to test which are the results with different kinds of functions such as bezier curves or a function of high order to produce a smoother transitions between two connected segments to avoid weight based artifacts. The algorithms depicted in this thesis are sequential due to the nature of our implementation as a **Maya** plug-in; an alternative will be a parallelized version on **CUDA** to improve its performance. The distribution function of the extra weight in our skinning algorithm is linear; different distribution functions may lead to different twist behaviors. Our skinning method is always plugged in its implementation; we believe that a more efficient method will achieve lower processing times, in a similar fashion as [22] that plugs and unplugs the method when a twist rotation is detected. An interesting topic to explore in our deformation algorithm is the possibility of implementing a

self-intersection mechanism and dynamics to improve the realism in the deformations. The main problem we saw is the extra computation that a dynamics module would imply; pushing the processing times farther from real time deformations. A method that approximates the dynamic behavior based on pre-computed data will be a viable option. Another interesting topic to explore as future work will be a volume preservation algorithm with an error of 0% or closer. As we have shown in 4.4, the volume loss of our deformation algorithm is low but it can be perfected with a volume-correction algorithm such as [44] that works as an extra module of a deformation algorithm.

6.3 Conclusions.

The main objective of this thesis was to create a set of tools that can work together to rig an arbitrary input model, and to make the whole rigging process easier to people with a low level of knowledge about the process. With this objective in mind we developed our method as a sequence of algorithms that can be stopped in different points of the process; or can take as an input rigged models that were not created with our method. Specifically our method can be subdivided into three sub-process: *Skeleton extraction and adjustment of a template skeleton* (to create a rig based on a template animation skeleton), *Segmentation and weight assign* and *Skinning method*, and can be used as a single process or in sequence.
Our contributions in the develop of this thesis are in a nutshell:

- We successfully created a *skeleton extraction and adjustment method* that can be used to rig a mesh in an arbitrary pose; using user defined markers while other methods are restricted to the ideal **T-pose** in human-like input meshes.

- To work along with our rig method we have developed a *weight assign* algorithm based on a novel segmentation algorithm, as its core. The segmentation is important because is a concept that simplifies and makes easier the weight assignment process, while other algorithms depend on minimization algorithms, or uses the weights assigned by other methods as a starting point. A good segmentation can be useful even for digital $3D$ artist as a base to *paint* influence weights on a desired model.

- The widely-used *LBS* algorithm has the well-known *candy wrapper* artifact, and to overcome this problem we developed a skinning algorithm based on *LBS*. Our algorithm can handle advance deformations (twists over a link greater than 180°), without volume loss or unrealistic artifacts; it's not dependent on examples using weights generated for a *LBS* deformation scheme and generates automatically the extra weights needed.

- Additionally, we have developed an animation tool that can load an animation file with a different skeleton, and adapt the data to a rigged mesh with some minor restrictions.

Our method was developed entirely in **Autodesk Maya** as a plug-in in ANSI C++; with the objective of making the diffusion around the animation community easier, independently of the hardware used. Although the project was made using the Visual C++ compiler of Microsoft; with some changes a port to Mac, Linux or another operating system that support Maya and a C++ compiler will be possible. Although the development in such a sophisticated platform was sometimes a challenge, our implementation was beneficed by a robust framework with solid math libraries. By teaming up our method of automatic rigging with the best animation software in the industry, we have achieved our goal of creating a method that can help a wide range of users with or without previous knowledge that need to rig and animate a three dimensional model with minimum interaction and low adjustment parameter number.

We conclude this thesis with a wider vision about the animation area, the problems faced in this thesis are far from being completely solved. The methods proposed in this work have too much room left to be improved. By its nature, computer animation is a fascinating field with a bright future that grows along not only with the computer hardware power; it is a multidisciplinary field that incorporates mathematics, dynamics, programming languages, and art. All this features make it one of the most demanded and interesting research fields up to date and in the future.

Bibliography

[1] ACCAD - Motion Capture Lab. `http://accad.osu.edu/research/mocap/mocap_data.htm`.

[2] Grégoire Aujay, Franck Hétroy, Francis Lazarus, and Christine Depraz. Harmonic skeleton for realistic character animation. In *Proceedings of the 2007 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 151–160. Eurographics Association, 2007.

[3] Ilya Baran and Jovan Popović. Automatic rigging and animation of 3d characters. In *SIGGRAPH '07: ACM SIGGRAPH 2007 papers*, page 72, New York, NY, USA, 2007. ACM.

[4] Gaurav Bharaj, Thorsten Thormählen, Hans-Peter Seidel, and Christian Theobalt. Automatically rigging multi-component characters. In *Computer Graphics Forum*, volume 31, pages 755–764. Wiley Online Library, 2012.

[5] Nicu D. Cornea, Deborah Silver, and Patrick Min. Curve-skeleton properties, applications, and algorithms. *IEEE Transactions on Visualization and Computer Graphics*, 13(3):530–548, 2007.

[6] Edilson de Aguiar, Christian Theobalt, Sebastian Thrun, and Hans-Peter Seidel. Automatic conversion of mesh animations into skeleton-based animations. *Comput. Graph. Forum*, 27(2):389–397, 2008.

[7] Tony Derose and Mark Meyer. Harmonic coordinates. In *Pixar Technical Memo 06-02, Pixar Animation Studios*, 2006.

[8] Olivier Dionne and Martin de Lasa. Geodesic voxel binding for production character meshes. In *Proceedings of the 12th ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '13, pages 173–180, New York, NY, USA, 2013. ACM.

[9] Michael S. Floater and Kai Hormann. Surface parameterization: a tutorial and survey. In *In Advances in Multiresolution for Geometric Modelling*, pages 157–186. Springer, 2005.

[10] David Gould. *Complete Maya Programming: An extensive guide to MEL and C++ API*. Morgan Kaufmann, 2003.

[11] David Jacka, Ashley Reid, and Bruce Merry. A comparison of linear skinning techniques for character animation. In *In Afrigraph*, pages 177–186. ACM, 2007.

[12] Alec Jacobson, Ilya Baran, Ladislav Kavan, Jovan Popović, and Olga Sorkine. Fast automatic skinning transformations. *ACM Transactions on Graphics (TOG)*, 31(4):77, 2012.

[13] Alec Jacobson, Ilya Baran, Jovan Popovic, and Olga Sorkine. Bounded biharmonic weights for real-time deformation. *ACM Trans. Graph.*, 30(4):78, 2011.

[14] Doug L. James and Christopher D. Twigg. Skinning mesh animations. In *ACM SIGGRAPH 2005 Papers*, SIGGRAPH '05, pages 399–407, New York, NY, USA, 2005. ACM.

[15] Pushkar Joshi, Mark Meyer, Tony DeRose, Brian Green, and Tom Sanocki. Harmonic coordinates for character articulation. In *ACM Transactions on Graphics (TOG)*, volume 26, page 71. ACM, 2007.

[16] Tao Ju, Scott Schaefer, and Joe Warren. Mean value coordinates for closed triangular meshes. *ACM Trans. Graph.*, 24(3):561–566, July 2005.

[17] Tao Ju, Qian-Yi Zhou, Michiel van de Panne, Daniel Cohen-Or, and Ulrich Neumann. Reusable skinning templates using cage-based deformations. *ACM Trans. Graph.*, 27(5):122:1–122:10, December 2008.

[18] Karan Singh Karan. Skinning characters using surface-oriented free-form deformations. In *In Graphics Interface 2000*, pages 35–42, 2000.

[19] Sagi Katz, George Leifman, and Ayellet Tal. Mesh segmentation using feature point and core extraction. *The Visual Computer*, 21(8-10):649–658, 2005.

[20] Ladislav Kavan, Steven Collins, and Carol O'Sullivan. Automatic linearization of nonlinear skinning. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, pages 49–56. ACM, 2009.

[21] Ladislav Kavan, Steven Collins, Jiří Žára, and Carol O'Sullivan. Geometric skinning with approximate dual quaternion blending. *ACM Trans. Graph.*, 27(4):105:1–105:23, November 2008.

[22] Ladislav Kavan and Olga Sorkine. Elasticity-inspired deformers for character articulation. *ACM Transactions on Graphics (TOG)*, 31(6):196, 2012.

[23] Ladislav Kavan and Jiří Žára. Spherical blend skinning: a real-time deformation of articulated models. In *Proceedings of the 2005 symposium on Interactive 3D graphics and games*, I3D '05, pages 9–16, New York, NY, USA, 2005. ACM.

[24] J. P. Lewis, Matt Cordner, and Nickson Fong. Pose space deformation: a unified approach to shape interpolation and skeleton-driven deformation. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '00, pages 165–172, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[25] Yaron Lipman, Johannes Kopf, Daniel Cohen-Or, and David Levin. Gpu-assisted positive mean value coordinates for mesh deformations. In *Proceedings of the fifth Eurographics symposium on Geometry processing*, SGP '07, pages 117–123, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association.

[26] Christophe Lohou and Gilles Bertrand. A new 3d 12-subiteration thinning algorithm based on p-simple points. *Discrete Appl. Math.*, 139(1-3):171–195, 2004.

[27] Bruce Merry, Patrick Marais, and James Gain. Animation space: A truly linear framework for character animation. *ACM Trans. Graph.*, 25(4):1400–1423, October 2006.

[28] Alex Mohr and Michael Gleicher. Building efficient, accurate character skins from examples. In *ACM SIGGRAPH 2003 Papers*, SIGGRAPH '03, pages 562–568, New York, NY, USA, 2003. ACM.

[29] Alex Mohr and Michael Gleicher. Deformation sensitive decimation. *Technical Report*, 2003.

[30] Alex Mohr, Luke Tokheim, and Michael Gleicher. Direct manipulation of interactive character skins. In *Proceedings of the 2003 symposium on Interactive 3D graphics*, I3D '03, pages 27–30, New York, NY, USA, 2003. ACM.

[31] C.J. Ogyar, A.J. Rueda, R.J. Segura, and F.R. Feito. Fast and simple hardware accelerated voxelizations using simplicial coverings. *The Visual Computer*, 23(8):535–543, 2007.

[32] Kálmán Palágyi and Attila Kuba. A 3d 6-subiteration thinning algorithm for extracting medial lines. *Pattern Recogn. Lett.*, 19(7):613–627, 1998.

[33] Kálmán Palágyi, Erich Sorantin, Emese Balogh, Attila Kuba, Csongor Halmai, Balázs Erdohelyi, and Klaus Hausegger. A sequential 3d thinning algorithm and its medical applications. In *IPMI '01: Proceedings of the 17th International Conference on Information Processing in Medical Imaging*, pages 409–415, London, UK, 2001. Springer-Verlag.

[34] JunJun Pan, Xiaosong Yang, Xin Xie, Philip Willis, and Jian J. Zhang. Automatic rigging for animation characters with 3d silhouette. *Comput. Animat. Virtual Worlds*, 20(2/3):121–131, 2009.

[35] Comparative Study Rafael, Rafael J. Segura, and Francisco R. Feito. Algorithms to test ray-triangle intersection. In *Journal of WSCG*, pages 200–1, 2001.

[36] Jorge E. Ramirez, Xavier Lligadas, and Antonio Susin. Automatic adjustment of rigs to extracted skeletons. In *AMDO '08: Proceedings of the 5th international conference on Articulated Motion and Deformable Objects*, pages 409–418, Berlin, Heidelberg, 2008. Springer-Verlag.

[37] Jorge E. Ramirez, Xavier Lligadas, and Antonio Susin. Adjusting animation rigs to human-like 3d models. In *AMDO '10: Proceedings of the 6th international conference on Articulated Motion and Deformable Objects*, pages 300–310, Berlin, Heidelberg, 2010. Springer-Verlag.

[38] Jorge E. Ramirez and Antonio Susin. Segmentation based skinning. *Computer Animation and Virtual Worlds*, 2015. accepted for publication 2015.

[39] Damien Rohmer, Stefanie Hahmann, and Marie-Paule Cani. Exact volume preserving skinning with shape control. In *Proceedings of the 2009 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '09, pages 83–92, New York, NY, USA, 2009. ACM.

[40] Ariel Shamir. A survey on mesh segmentation techniques. *Computer Graphics Forum*, 27(6):1539–1556, 2008.

[41] Justin Slick. 7 common modeling techniques for film and games. `http://3d.about.com/od/3d-101-The-Basics/a/Introduction-To-3d-Modeling-Techniques.htm`. Accessed: 2015-11-12.

[42] Peter-Pike J. Sloan, Charles F. Rose, III, and Michael F. Cohen. Shape by example. In *Proceedings of the 2001 symposium on Interactive 3D graphics*, I3D '01, pages 135–143, New York, NY, USA, 2001. ACM.

[43] Tony Tung and Francis Schmitt. F.: Augmented reeb graphs for content-based retrieval of 3d mesh models. In *In SMI 04: Proceedings of the Shape Modeling International 2004 (SMI04*, pages 157–166, 2004.

[44] W. von Funck, H. Theisel, and H.P. Seidel. Volume-preserving Mesh Skinning. *Proceedings of Vision, Modeling, and Visualization 2008*, page 409, 2008.

[45] Lawson Wade. *Automated generation of control skeletons for use in animation*. PhD thesis, 2000. Adviser-Parent, Richard E.

[46] Xiaohuan Corina Wang and Cary Phillips. Multi-weight enveloping: least-squares approximation techniques for skin animation. In *Proceedings of the 2002 ACM SIGGRAPH/Eurographics symposium on Computer animation*, SCA '02, pages 129–138, New York, NY, USA, 2002. ACM.

[47] Ofir Weber, Olga Sorkine, Yaron Lipman, and Craig Gotsman. Context-aware skeletal shape deformation. In *Computer Graphics Forum*, volume 26, pages 265–274. Wiley Online Library, 2007.

[48] Fu-Che Wu, Wan-Chun Ma, Ping-Chou Liou, Rung-Huei Laing, and Ming Ouhyoung. Skeleton extraction of 3d objects with visible repulsive force. In *Computer Graphics Workshop*, pages 124–131, 2003.

[49] X. S. Yang and Jian J. Zhang. Realistic skeleton driven skin deformation. In *Proceedings of the 2005 international conference on Computational Science and Its Applications - Volume Part III*, ICCSA'05, pages 1109–1118, Berlin, Heidelberg, 2005. Springer-Verlag.

[50] Xiaosong Yang and Jian J. Zhang. Stretch it - realistic smooth skinning. In *Proceedings of the International Conference on Computer Graphics, Imaging and Visualisation*, CGIV '06, pages 323–328, Washington, DC, USA, 2006. IEEE Computer Society.